Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα Υπολογιστών

3η Εργαστηριακή Άσκηση

Συγχρονισμός

Ροή: Υ

Ομάδα: Α05

Μέλη: Μιχαήλ Σωτήρης *03113719*

Νικολαΐδης Ανδρέας *03113715*

Εξάμηνο: 7ο

Ακαδ. έτος: 2016-2017

Άσκηση 1.1 – : Συγχρονισμός σε υπάρχοντα κώδικα

Χρησιμοποιήσαμε το παρεχόμενο Makefile για να μεταγλωττίσουμε και να τρέξουμε το πρόγραμμα simplesync.c. Παρατηρήσαμε ότι για κάθε αρχείο .c χρησιμοποιείται η παράμετρος -pthread.

Ο κώδικας χρησιμοποιεί threads (νήματα). Άρα είναι απαραίτητο ο linker να το γνωρίζει ώστε να χρησιμοποιήσει τις ασφαλείς εκδόσεις συναρτήσεων της βιβλιοθήκης C ως προς το threading. Επίσης , παρατηρούμε ότι ενώ έχουμε ένα αρχείο με όνομα simplesync.c το Makefile εκτελεί δυο μεταγλωττίσεις για αυτό.

Παράγεται ένα με τη παράμετρο -DSYNC_MUTEX και ένα με τη παράμετρο -DSYNC_ATOMIC. Και τα δύο χρησιμοποιούν τον ίδιο κώδικα simplesync.c για να παράξουν ένα .o αρχείο το καθένα και από τα .o αρχεία φτιάχνονται τα εκτελέσιμα simplesync-atomic και simplesync-mutex.

```
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 -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
```

Επεκτείναμε, όπως μας ζητείται στην άσκηση, τον κώδικα του simplesync.c ώστε ο ίδιος να υποστηρίζει στην περίπτωση του simplesync-mutex συγχρονισμό με χρήση POSIX mutexes και στη περίπτωση του simplesync-atomic να υποστηρίζει συγχρονισμό με χρήση ατομικών λειτουργιών του GCC.

Πιο κάτω ακολουθεί ο κώδικας και αποτέλεσμα εξόδου εκτελέσιμων simplesync-mutex και simplesync-atomic.

Κώδικας:

```
/*
 * 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


pthread_mutex_t new_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;
	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);
		else {
			pthread_mutex_lock(&new_mutex);
/* The mutex object is locked by calling pthread_mutex_lock(). If the mutex is already
 * locked, the calling thread blocks until the mutex becomes available.
 * This operation returns with the mutex object referenced by mutex
 * in the locked state with the calling thread as its owner.
 */
			++(*ip);
			pthread_mutex_unlock(&new_mutex);

/* The pthread_mutex_unlock() function releases the mutex object referenced by mutex. */
		}
	} // end of for

	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);
```

```
/* An operation acting on shared memory is atomic if it completes in a single step relative to other threads.
 * When an atomic store is performed on a shared variable, no other thread can observe the modification
 * half-complete. When an atomic load is performed on a shared variable, it reads the entire value as it
 * appeared at a single moment in time. Non-atomic loads and stores do not make those guarantees.
 */
                else {
                        pthread_mutex_lock(&new_mutex);
                        --(*ip);
                        pthread_mutex_unlock(&new_mutex);
                }
        } // end of for

        fprintf(stderr, "Done decreasing variable.\n");

        return NULL;
}


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

        int val, ret, ok;
        pthread_t t1, t2;

        val = 0; // initial value

        // creates 2 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 the 2 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");

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

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

        return ok;
}
```

ΕΞΟΔΟΣ ΕΚΤΕΛΕΣΗΣ

Αποτέλεσμα εξόδου εκτέλεσης αρχείου simplesync-mutex

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

Αποτέλεσμα εξόδου εκτέλεσης αρχείου simplesync-atomic

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

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 στις περιπτώσεις που έχουμε συγχρονισμό ο χρόνος που απαιτείται για να εκτελεστεί το πρόγραμμα είναι μεγαλύτερος από τη περίπτωση που δεν έχουμε συγχρονισμό. Αυτό οφείλεται στο γεγονός ότι χωρίς την ύπαρξη του συγχρονισμού πολλές πράξεις δεν εκτελούνται και έτσι απαιτείται λιγότερος χρόνος για την εκτέλεση του προγράμματος. Αυτό όμως έχει και ως επίπτωση το αποτέλεσμα να είναι κατά μεγάλη πιθανότητα λάθος.

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

   Η μέθοδος ατομικών λειτουργιών είναι πιο γρήγορη από τη μέθοδο POSIX mutexes. Αυτό οφείλεται στο γεγονός ότι οι ατομικές λειτουργίες προσπαθούν να εκτελεστούν συνέχεια μέχρι να τα καταφέρουν ενώ με POSIX mutexes, οι διεργασίες αναστέλλουν την λειτουργίας τους όσο το κύριο τμήμα είναι κατειλημμένο.

3. ***Σε ποιές 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.***

   Χρησιμοποιώντας τις παραμέτρους -S και -g στις εντολές :
   gcc -O2 -S -g -DSYNC_ATOMIC -c simplesync.c
   gcc -c -g -Wa,-a,-ad -DSYNC_ATOMIC simplesync.c > atomic.lst

   όπου :
   -S
   Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

   -Wa,option
   Pass option as an option to the assembler. If option contains commas, it is split into multiple options at the commas.

-a
defaults to -ahls

-ad
omit debugging directives

παίρνουμε στο αρχείο atomic.lst τον ενδιάμεσο κώδικα Assembly μαζί με πληροφορίες γραμμών πηγαίου κώδικα. Στο αρχείο που παράγεται, ανάμεσα σε άλλα, ξεχωρίζουμε το ακόλουθο τμήμα :

```
52:simplesync.c  ****                   if (USE_ATOMIC_OPS)
53:simplesync.c  ****                           __sync_add_and_fetch(ip, 1);
40                      .loc 1 53 0
41 0030 8B45F0                  movl    -16(%ebp), %eax
42 0033 F0830001                lock addl       $1, (%eax)
51:simplesync.c  ****           for (i = 0; i < N; i++) {
43                      .loc 1 51 0
44 0037 8345F401                addl    $1, -12(%ebp)
45              .L2:
51:simplesync.c  ****           for (i = 0; i < N; i++) {
46                      .loc 1 51 0 is_stmt 0 discriminator 1
47 003b 817DF47F                cmpl    $9999999, -12(%ebp)

GAS LISTING /tmp/ccYtX8aK.s                             page 3


47      969800
48 0042 7EEC                    jle     .L3
```

Από το παραπάνω τμήμα συμπεραίνουμε πως οι ατομικές λειτουργίες του GCC στην αρχιτεκτονική μεταφράζονται στις εξής εντολές του επεξεργαστή (γραμμή 42):

lock addl $1, (%eax)

Αντίστοιχα , για τη μείωση ξεχωρίζουμε το τμήμα :

```
80:simplesync.c  ****                   if (USE_ATOMIC_OPS)
 81:simplesync.c  ****                           __sync_sub_and_fetch(ip, 1);
101                     .loc 1 81 0
102 0092 8B45F0                 movl    -16(%ebp), %eax
103 0095 F0832801               lock subl       $1, (%eax)
 79:simplesync.c  ****          for (i = 0; i < N; i++) {
104                     .loc 1 79 0
105 0099 8345F401               addl    $1, -12(%ebp)
106             .L6:
 79:simplesync.c  ****          for (i = 0; i < N; i++) {
107                     .loc 1 79 0 is_stmt 0 discriminator 1
108 009d 817DF47F               cmpl    $9999999, -12(%ebp)
108     969800
109 00a4 7EEC                   jle     .L7
```

στο οποίο η εντολή του επεξεργαστή (γραμμή 103) είναι η εξής:
lock subl $1, (%eax)

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.***

gcc -O2 -S -g -DSYNC_MUTEX -c simplesync.c
gcc -c -g -Wa,-a,-ad -DSYNC_MUTEX simplesync.c > mutex.lst

Παίρνουμε από το mutex.lst τις εντολές στις οποίες μεταφράζεται η χρήση POSIX mutex, για την αύξηση :

```
 55:simplesync.c  ****          pthread_mutex_lock(&new_mutex);
 40                     .loc 1 55 0
 41 0030 83EC0C                 subl    $12, %esp
 42 0033 68000000               pushl   $new_mutex
 42     00
 43 0038 E8FCFFFF               call    pthread_mutex_lock
 43     FF
 44 003d 83C410                 addl    $16, %esp
 61:simplesync.c  ****                          ++(*ip);
 45                     .loc 1 61 0
 46 0040 8B45F0                 movl    -16(%ebp), %eax
 47 0043 8B00                   movl    (%eax), %eax
```

και για τη μείωση :

```
88:simplesync.c  ****                          pthread_mutex_lock(&new_mutex);
 114                    .loc 1 88 0
 115 00b8 83EC0C              subl    $12, %esp
 116 00bb 68000000            pushl   $new_mutex
 116     00
 117 00c0 E8FCFFFF            call    pthread_mutex_lock
 117     FF
 118 00c5 83C410              addl    $16, %esp
 89:simplesync.c  ****                          --(*ip);
 119                    .loc 1 89 0
 120 00c8 8B45F0              movl    -16(%ebp), %eax
 121 00cb 8B00                movl    (%eax), %eax
 122 00cd 8D50FF              leal    -1(%eax), %edx
 123 00d0 8B45F0              movl    -16(%ebp), %eax
 124 00d3 8910                movl    %edx, (%eax)
```

Άσκηση 1.2 – Παράλληλος υπολογισμός του συνόλου Mandelbrot

Κώδικας:

```
/*
 * 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 <semaphore.h>
#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

/***************************
 * Compile-time parameters *
 ***************************/
int n;

sem_t *mysem; // The <semaphore.h> header defines the sem_t type, used in performing semaphore operations.

int y_chars = 50; // Output at the terminal is
int x_chars = 90; // x_chars wide by y_chars long

int X[50][90];


// The part of the complex plane to be drawn:
double xmin = -1.8, xmax = 1.0; // upper left corner is (xmin, ymax),
double ymin = -1.0, ymax = 1.0; // lower right corner is (xmax, ymin)


double xstep; // Every character in the final output is
double ystep; // xstep x ystep units wide on the complex plane.
```

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

	l = strtol(s, &endp, 10); //converts the string s to a long integer according to base 10
	if (s != endp && *endp == '\0') {
		*val = l;
		return 0;
	   // sto endp grafonte oi lathos xaraktireS p den mporoun na ginoun sti vasi
	   // opote an einai \0 simenei oti egine  olokliro to s integer xoris provlima
	} else  return -1;
}

void compute_mandel_line(int line) // computes a line of output
{ 	// as an array of x_char color values
	double x, y; //x and y traverse the complex plane.

	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);
		X[line][n] = val;
	}
}
```

```
void output_mandel_line(int fd, int n1)
{ // This function outputs an array of x_char color values to a 256-color xterm
	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, X[n1][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);
	}
}

// each new thread invokes this function :
void *thread_start_fn(void *arg) // arg is a pointer to where this thread's id is kept
{
	volatile int *n1 = arg; //pointer to a volatile int , initialized to point where this thread's id
is kept
	int i;

     /* We know arg points to an instance of thread_info_struct */
     for (i = *n1; i < y_chars; i += n) {  // each thread calculates its line (its id) and all of (id+n)
other lines
	compute_mandel_line(i);       // compute 1 line at a time , the i-line
		sem_wait(&mysem[*n1]); // and LOCK the semaphor at address &mysem[i] IN
ORDER TO BE THE ONLY THREAD TO EXECUTE output_mandel_line() NEXT.

		// sem_wait(sem) decrements (locks) the semaphore pointed to by sem.
		// If the semaphore's value is greater than zero, then the decrement proceeds,
and the function returns, immediately.
		// If the semaphore currently has the value zero, then the call blocks until :
```

```
                // either it becomes possible to perform the decrement (i.e., the semaphore
value rises above zero),
                // or a signal handler interrupts the call.

                // fprintf(stderr, "eimai o %d \n",m);

                output_mandel_line(1, i);

                // each semaphor posts the semaphor of the next thread. the semaphor of the
N-th thread posts the semaphor of the 1st thread.
                if (*n1==n-1)   sem_post(&mysem[0]); // the very last semaphor posts the
semaphor at address &mysem[0]
                else            sem_post(&mysem[*n1 + 1]); // all other semaphors post the
semaphor of their next thread, so that the next thread wakes up.

                // sem_post() increments (unlocks) the semaphore pointed to by its argument.
                // If the semaphore's value consequently becomes greater than zero, then
another process or
                // thread blocked in a sem_wait(3) call will be woken up and proceed to lock the
semaphore.
                // sem_post() returns 0 on success; on error, the value of the semaphore is left
unchanged,
                //-1 is returned, and errno is set to indicate the error.

        }
    return NULL;
}

int main(int argc, char *argv[])
{
        int i, ret ;
        pthread_t *thrid;
        int *no;

        // CKECK INPUT
        if (argc != 2) { fprintf(stderr, "wrong arguments\nUsage: ./mandel numberofthreads
\nexiting !"); exit(1); }
```

```
// convert string argv[1] to long int and save to variable n.
if (safe_atoi(argv[1],&n)) // on success , zero is returned
  { fprintf(stderr, "`%s'is not a number\n", argv[1]); exit(1); }

thrid = malloc(n * sizeof(pthread_t));          // thread space
no = malloc(n*sizeof(int));              // no[N] , one int for each id i of a thread

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 (line = 0; line < y_chars; line++) {
//      compute_and_output_mandel_line(1, line);
//}

mysem = malloc(n*sizeof(sem_t));    // array mysem[N]  :  allocate space for N
semaphores

// initialize
sem_init(&mysem[0],0,1);                        // semaphore 0 : allows 1 thread inside
for(i = 1; i<n; i++ ) sem_init(&mysem[i],0,0); // semaphores 1 to n : allow 0 threads
inside at the beginning (until a semaphore posts)

// The 1st argument indicates that sem_init() initializes the unnamed semaphore at
&mysem[i].
// The 2nd argument indicates : 0 for semaphore between the threads of a process, else
is shared between processes.
// The 3rd argument specifies the initial value for the semaphore : how many threads to
allow at the beginning (until a semaphore posts)

for( i = 0; i<n; i++ ){  // for each thread (out of N)
        no[i] = i;          // number-name of current thread is current i

        // spawn this new thread . it will invoke thread_start_fn with argument &no[i]
        ret = pthread_create(&thrid[i], NULL, thread_start_fn, &no[i]);

        if (ret) { fprintf(stderr,"pthread_create() returned %d \n",ret); exit(1); }
}
```

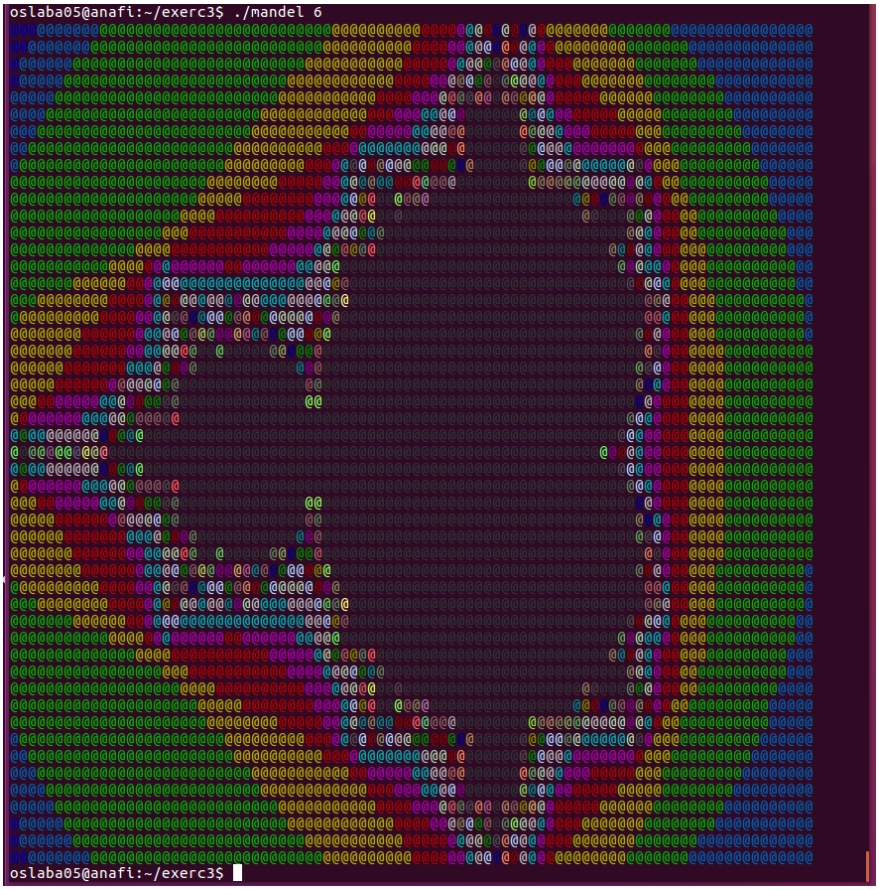
```
	// wait for all threads to terminate ( if already terminated it returns immediately )
	for (i = 0; i < n; i++) {
		ret = pthread_join(thrid[i], NULL);
		if (ret) { fprintf(stderr,"pthread_join() returned %d \n",ret); exit(1); }
	}

	// destroy all semaphores
	for( i = 0; i<n; i++ ) sem_destroy(&mysem[i]);

	// reset terminal color to initial setting
	reset_xterm_color(1);

	return 0;
}
```

ΕΞΟΔΟΣ ΕΚΤΕΛΕΣΗΣ

Αποτέλεσμα εξόδου εκτέλεσης αρχείου mandel με διάφορες εισόδους

```
:~$ ./mandel
wrong arguments
Usage: ./mandel numberofthreads
exiting !
:~$  ./mandel 4 5
wrong arguments
Usage: ./mandel numberofthreads
exiting !
```

:~$ ./mandel 6



Ερωτήσεις:

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

   Χρειάζονται τόσοι σημαφόροι, όσα είναι τα νήματα, ώστε να μπορούμε να ελέγχουμε πότε θα έχει πρόσβαση το κάθε νήμα στο κύριο τμήμα. Αυτό έχει νόημα καθώς κάθε νήμα μπορεί να εκτελεί τις εντολές του “παράλληλα” με τα άλλα, και μόνο όταν πρόκειται να χρησιμοποιήσει κάποιο κοινό πόρο (μνήμη , οθόνη , κτλ) μπαίνει στη διαδικασία να μειώσει (κλειδώσει) τον δικό του σημαφόρο , παίρνοντας τον έλεγχο και αποτρέποντας άλλα νήματα να κάνουν την ίδια κίνηση μέχρι αυτό να δώσει τον έλεγχο στον σημαφόρο του επόμενου στη σειρά νήματος.

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.***

   Για την σειριακή εκτέλεση
   real 0m0.946s
   user 0m0.932s
   sys 0m0.008s

   Για την παράλληλη εκτέλεση
   real 0m0.509s
   user 0m0.904s
   sys 0m0.016s

3. ***Το παράλληλο πρόγραμμα που φτιάξατε, εμφανίζει επιτάχυνση; Αν όχι, γιατί; Τι πρόβλημα υπάρχει στο σχήμα συγχρονισμού που έχετε υλοποιήσει; Υπόδειξη: Πόσο μεγάλο είναι το κρίσιμο τμήμα; Χρειάζεται να περιέχει και τη φάση υπολογισμού και τη φάση εξόδου κάθε γραμμής που παράγεται;***

   Παρατηρούμε ότι η παράλληλη εκτέλεση απαιτεί λιγότερο χρόνο όπως είναι και λογικό αφού το κάθε νήμα δεν περιμένει να τελειώσει το προηγούμενό του για να ξεκινήσει. Ωστόσο το κρίσιμο τμήμα αποτελείται μόνο από τη φάση εξόδου της κάθε γραμμής, αφού ο υπολογισμός της κάθε γραμμής δεν εξαρτάται από τον υπολογισμό της προηγούμενης γραμμής και έτσι το κρίσιμο τμήμα δεν είναι μεγάλο σε μέγεθος. Έτσι, λόγω του μεγέθους υπάρχει επιτάχυνση.

4. ***Τι συμβαίνει στο τερματικό αν πατήσετε Ctrl-C ενώ το πρόγραμμα εκτελείται; σε τι κατάσταση αφήνεται, όσον αφορά το χρώμα των γραμμάτων; Πώς θα μπορούσατε να επεκτείνετε το mandel.c σας ώστε να εξασφαλίσετε ότι ακόμη κι αν ο χρήστης πατήσει Ctrl-C, το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του;***

   Όταν πατήσουμε Ctrl-C ενώ το πρόγραμμα εκτελείται, στέλνουμε σήμα -KILL στη διεργασία και το πρόγραμμα τερματίζει. Επομένως η αλλαγή των χρωμάτων των χαρακτήρων δεν επαναφέρεται και παραμένει το χρώμα που τέθηκε τελευταίο και συνεπώς παραμένει χρωματισμένο το terminal. Για να αντιμετωπίσουμε το πρόβλημα, το πρόγραμμα θα μπορούσε να διαχειρίζεται το σήμα Ctrl-C με τη συνάρτηση χειρισμού σήματος sighandler() στην οποία θα υπάρχουν οι εντολές reset_xterm_color(1) και η exit(1). Έτσι όταν θα πατήσουμε Ctrl-C, θα ενεργοποιούμε το sighandler() και αυτό θα επαναφέρει το αρχικό χρώμα και μετά θα κάνει έξοδο προγράμματος.

Άσκηση 1.3– Επίλυση προβλήματος συγχρονισμού

Κώδικας:

```
/*
 * kgarten.c
 *
 * A kindergarten simulator.
 * Bad things happen if teachers and children
 * are not synchronized properly.
 *
 *
 * Author:
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 *
 * Additional Authors:
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 * Anastassios Nanos <ananos@cslab.ece.ntua.gr>
 * Operating Systems course, ECE, NTUA
 *
 */

#include <time.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.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)
```

```
/* A virtual kindergarten */
struct kgarten_struct {

	/* Here you may define any mutexes / condition variables / other variables  you may
need.*/
	pthread_cond_t cond;		/* a condition variable of kindergarten*/


	/* You may NOT modify or use anything in the structure below this */
	/* point. They are only meant to be used by the framework code, for verification. */
	int vt;
	int vc;
	int ratio;

	pthread_mutex_t mutex;
};

/* A (distinct) instance of this structure
* is passed to each thread
*/
struct thread_info_struct {
	pthread_t tid; /* POSIX thread id, as returned by the library */

	struct kgarten_struct *kg;
	int is_child;  /* Nonzero if this thread simulates children, zero otherwise */

	int thrid;     /* Application-defined thread id */
	int thrcnt;
	unsigned int rseed; /* for random number generator*/
};

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

	l = strtol(s, &endp, 10); /* convert string s to long int according to base 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 usage(char *argv0)
{
	fprintf(stderr, "Usage: %s thread_count child_threads c_t_ratio\n\n"
		"Exactly two argument required:\n"
		"    thread_count: Total number of threads to create.\n"
		"    child_threads: The number of threads simulating children.\n"
		"    c_t_ratio: The allowed ratio of children to teachers.\n\n",
		argv0);
	exit(1);
}

void bad_thing(int thrid, int children, int teachers)
{
	int thing, sex;
	int namecnt, nameidx;
	char *name, *p;
	char buf[1024];

	char *things[] = {
		"Little %s put %s finger in the wall outlet and got electrocuted!",
		"Little %s fell off the slide and broke %s head!",
		"Little %s was playing with matches and lit %s hair on fire!",
		"Little %s drank a bottle of acid with %s lunch!",
		"Little %s caught %s hand in the paper shredder!",
		"Little %s wrestled with a stray dog and it bit %s finger off!"
	};
	char *boys[] = {
		"George", "John", "Nick", "Jim", "Constantine",
		"Chris", "Peter", "Paul", "Steve", "Billy", "Mike",
		"Vangelis", "Antony"
	};
```

```
	char *girls[] = {
		"Maria", "Irene", "Christina", "Helena", "Georgia", "Olga",
		"Sophie", "Joanna", "Zoe", "Catherine", "Marina", "Stella",
		"Vicky", "Jenny"
	};

	thing = rand() % 4;
	sex = rand() % 2;

	namecnt = sex ? sizeof(boys)/sizeof(boys[0]) :		sizeof(girls)/sizeof(girls[0]);
	nameidx = rand() % namecnt;
	name = sex ? boys[nameidx] : girls[nameidx];

	p = buf;
	p += sprintf(p, "*** Thread %d: Oh no! ", thrid);
	p += sprintf(p, things[thing], name, sex ? "his" : "her");
	p += sprintf(p, "\n*** Why were there only %d teachers for %d	children?!\n",
		teachers, children);

	/* Output everything in a single atomic call */
	printf("%s", buf);
}

void child_enter(struct thread_info_struct *thr) /* ON CHILD ENTER */
{
	if (!thr->is_child) { fprintf(stderr, "Internal error: %s called for a Teacher
thread.\n",__func__); exit(1); }


	/* CRITICAL SECTION START */

	pthread_mutex_lock(&thr->kg->mutex);

	/* while number of children equals EXACTLY the ratio , NO MORE CHILDREN CAN ENTER
, must wait */
	while((((thr->kg->vc)/(thr->kg->ratio))>=(thr->kg->vt))||((thr->kg->vt)==0))
		pthread_cond_wait(&thr->kg->cond,&thr->kg->mutex);

	++(thr->kg->vc); /* there is space for a new child , A CHILD ENTERS */
	fprintf(stderr, "THREAD %d: CHILD ENTER\n", thr->thrid);
	pthread_mutex_unlock(&thr->kg->mutex);

	/* CRITICAL SECTION END */
}
```

```
void child_exit(struct thread_info_struct *thr)  /* A CHILD CAN EXIT WHENEVER IT WANTS */
{
	if (!thr->is_child) {
		fprintf(stderr, "Internal error: %s called for a Teacher thread.\n", __func__);
		exit(1);
	}

	/* CRITICAL SECTION START */
	pthread_mutex_lock(&thr->kg->mutex);

	--(thr->kg->vc); /* child exits */

	pthread_cond_broadcast(&thr->kg->cond); /* broadcast message to others using same
cond that there is a space for a child by now ! */

	/* The pthread_cond_broadcast() function shall unblock all threads currently blocked on
the specified condition variable cond */
	/* If MORE THAN ONE thread is blocked on a condition variable, the scheduling policy
shall DETERMINE THE ORDER in which threads are unblocked */

	fprintf(stderr, "THREAD %d: CHILD EXIT\n", thr->thrid);

	pthread_mutex_unlock(&thr->kg->mutex);
	/* CRITICAL SECTION END */
}

void teacher_enter(struct thread_info_struct *thr) /* TEACHERS ENTER ANYTIME THEY WISH */
{
	if (thr->is_child) {
		fprintf(stderr, "Internal error: %s called for a Child thread.\n", __func__);
		exit(1);
	}

	/* CRITICAL SECTION START */
	pthread_mutex_lock(&thr->kg->mutex);

	++(thr->kg->vt); /* teachers increase */

	fprintf(stderr, "THREAD %d: TEACHER ENTER\n", thr->thrid);

	pthread_cond_broadcast(&thr->kg->cond); /* broadcast a message to others using
same
```

```
                                        cond that THERE IS SPACE FOR (RATIO)-MANY
CHILDREN TO ENTER */

        pthread_mutex_unlock(&thr->kg->mutex);
        /* CRITICAL SECTION END */
}

void teacher_exit(struct thread_info_struct *thr) /* ON TEACHER EXIT */
{
        if (thr->is_child) { fprintf(stderr, "Internal error: %s called for a Child        thread.\n",
__func__); exit(1); }

        /* CRITICAL SECTION START */
        pthread_mutex_lock(&thr->kg->mutex);

        /* assuming a teacher leaves , we have a NEW RATIO */
        /* while the new ratio is NOT VALID the teacher WAITS */
        while((((thr->kg->vt)-1)*(thr->kg->ratio))<(thr->kg->vc))
                pthread_cond_wait(&thr->kg->cond,&thr->kg->mutex);

        /* Upon new ratio BECOMES VALID , the teacher leaves safely */
        fprintf(stderr, "THREAD %d: TEACHER EXIT\n", thr->thrid);

        --(thr->kg->vt); /* decrease teachers (total) */

        pthread_mutex_unlock(&thr->kg->mutex);
        /* CRITICAL SECTION END */
}


/* Verify the state of the kindergarten. */
void verify(struct thread_info_struct *thr)
{
    struct kgarten_struct *kg = thr->kg;
    int t, c, r;

    c = kg->vc;
    t = kg->vt;
    r = kg->ratio;

    fprintf(stderr, "~VERIFY:       Thread %d: Teachers: %d, Children: %d\n", thr->thrid, t, c);
```

```
 if (c > t * r) {  /* if children more than  ( teachers X ratio )*/
            bad_thing(thr->thrid, c, t); /* a bad thing happens */
            exit(1);
      }
}

/* A single thread. It simulates either a teacher, or a child. */

void *thread_start_fn(void *arg)
{
        /* We know arg points to an instance of thread_info_struct */
        struct thread_info_struct *thr = arg; /* &thr[current i] */
        char *nstr;

        fprintf(stderr, "Thread %d of %d. START.\n", thr->thrid, thr->thrcnt);

        nstr = thr->is_child ? "Child" : "Teacher";
        for (;;) {
                fprintf(stderr, "Thread %d [%s]: Entering.\n", thr->thrid, nstr);

                if (thr->is_child)  child_enter(thr);           /* ENTER KG */
                else            teacher_enter(thr);

                fprintf(stderr, "Thread %d [%s]: Entered.\n", thr->thrid, nstr);

                /* We're inside the critical section, just sleep for a while. */
                /*usleep(rand_r(&thr->rseed) % 1000000 / (thr->is_child ? 10000 : 1)); */

                /* CRITICAL SECTION START */

        pthread_mutex_lock(&thr->kg->mutex);
                verify(thr);                                    /* VERIFY */
        pthread_mutex_unlock(&thr->kg->mutex);
                /* CRITICAL SECTION END */
                usleep(rand_r(&thr->rseed) % 1000000);

                fprintf(stderr, "Thread %d [%s]: Exiting.\n", thr->thrid, nstr);



                if (thr->is_child)  child_exit(thr);            /* EXIT KG */
                else            teacher_exit(thr);

                fprintf(stderr, "Thread %d [%s]: Exited.\n", thr->thrid, nstr);
```

```
		/* Sleep for a while before re-entering */
		/* usleep(rand_r(&thr->rseed) % 100000 * (thr->is_child ? 100 : 1)); */

		usleep(rand_r(&thr->rseed) % 100000);

		/* CRITICAL SECTION START */

	pthread_mutex_lock(&thr->kg->mutex);
		verify(thr);					/* VERIFY */
	pthread_mutex_unlock(&thr->kg->mutex);

		/* CRITICAL SECTION END */
	}

	fprintf(stderr, "Thread %d of %d. END.\n", thr->thrid, thr->thrcnt);

	return NULL;
}


int main(int argc, char *argv[])
{
	int i, ret, thrcnt, chldcnt, ratio;
	struct thread_info_struct *thr; /* a struct for each thread */
	struct kgarten_struct *kg; /* a single kg only */

	/* INPUT CHECK */
	if (argc != 4) { usage(argv[0]); exit(1) ; }

	if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0) {
	  fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
		exit(1);
	}
	if (safe_atoi(argv[2], &chldcnt)<0|| chldcnt < 0 || chldcnt > thrcnt) {
	fprintf(stderr, "`%s' is not valid for `child_threads'\n", argv[2]);
		exit(1);
	}
	if (safe_atoi(argv[3], &ratio) < 0 || ratio < 1) {
		fprintf(stderr, "`%s' is not valid for `c_t_ratio'\n", argv[3]);
		exit(1);
	}
```

```
        /* Initialize random number generator */
        srand(time(NULL));

        /*Initialize kindergarten  */
        kg = safe_malloc(sizeof(*kg));
        kg->vt = kg->vc = 0;
        kg->ratio = ratio;

        /* The pthread_mutex_init() function shall initialize the
         * mutex 1st arument with attributes specified by 2nd argument.
         *
         * If 2nd arg is NULL, the default mutex attributes are used;
         * the effect shall be the same as passing the address of a default mutex attributes
object.
         *
         * Upon successful initialization, mutex is UNLOCKED and 0 is returned.
         * */

        ret = pthread_mutex_init(&kg->mutex, NULL);
        if (ret) {  perror_pthread(ret, "pthread_mutex_init");  exit(1);  }

         /* Create threads */

        thr = safe_malloc(thrcnt * sizeof(*thr)); /* allocates array thr[thrcnt]*/

        for (i = 0; i < thrcnt; i++) {
                /* Initialize per-thread structure */
                thr[i].kg = kg;
                thr[i].thrid = i;
                thr[i].thrcnt = thrcnt;
                thr[i].is_child = (i < chldcnt);  /* firstly spawns all children and then all teachers*/
                thr[i].rseed = rand(); /* get a random seed for number generator*/

                /* Spawn new thread */
                ret = pthread_create(&thr[i].tid, NULL, thread_start_fn, &thr[i]);

                /* The pthread_create() function STARTS a new thread in the calling process. */
                /* The new thread starts execution by INVOKING thread_start_fn() */
                /*  &thr[i] is passed as the sole ARGUMENT of thread_strt_fn().*/

                if (ret) { perror_pthread(ret, "pthread_create");  exit(1);  }
        }
```

```
	/* Wait for all threads to terminate */
	for (i = 0; i < thrcnt; i++) {

		ret = pthread_join(thr[i].tid, NULL);

		/*The pthread_join() function WAITS for the thread specified by thread  to
TERMINATE. */
		/* If that thread has already terminated, then pthread_join() returns
immediately. */
		/* returns 0 on success*/

		if (ret) {
			perror_pthread(ret, "pthread_join");
			exit(1);
		}
	}

	printf("OK.\n");

	return 0;
}
```

ΕΞΟΔΟΣ ΕΚΤΕΛΕΣΗΣ

```
:~$ ./kgarten 8 3
Usage: ./kgarten thread_count child_threads c_t_ratio

Exactly two argument required:
   thread_count: Total number of threads to create.
   child_threads: The number of threads simulating children.
   c_t_ratio: The allowed ratio of children to teachers.

:~$ clear

:~$ ./kgarten 8 3
Usage: ./kgarten thread_count child_threads c_t_ratio

Exactly two argument required:
   thread_count: Total number of threads to create.
   child_threads: The number of threads simulating children.
   c_t_ratio: The allowed ratio of children to teachers.
```

(Δεν τερματίζει)

```
oslaba05@kefalonia:~/exerc3$ ./kgarten 15 10 3
Thread 6 of 15. START.
Thread 6 [Child]: Entering.
Thread 7 of 15. START.
Thread 7 [Child]: Entering.
Thread 8 of 15. START.
Thread 8 [Child]: Entering.
Thread 9 of 15. START.
Thread 9 [Child]: Entering.
Thread 10 of 15. START.
Thread 10 [Teacher]: Entering.
THREAD 10: TEACHER ENTER
Thread 10 [Teacher]: Entered.
~VERIFY:        Thread 10: Teachers: 1, Children: 0
THREAD 6: CHILD ENTER
Thread 6 [Child]: Entered.
~VERIFY:        Thread 6: Teachers: 1, Children: 1
THREAD 7: CHILD ENTER
Thread 7 [Child]: Entered.
~VERIFY:        Thread 7: Teachers: 1, Children: 2
THREAD 8: CHILD ENTER
Thread 8 [Child]: Entered.
~VERIFY:        Thread 8: Teachers: 1, Children: 3
Thread 11 of 15. START.
Thread 11 [Teacher]: Entering.
THREAD 11: TEACHER ENTER
Thread 11 [Teacher]: Entered.
~VERIFY:        Thread 11: Teachers: 2, Children: 3
THREAD 9: CHILD ENTER
Thread 9 [Child]: Entered.
~VERIFY:        Thread 9: Teachers: 2, Children: 4
Thread 12 of 15. START.
Thread 12 [Teacher]: Entering.
THREAD 12: TEACHER ENTER
Thread 12 [Teacher]: Entered.
~VERIFY:        Thread 12: Teachers: 3, Children: 4
```

Ερωτήσεις:

1. ***Έστω ότι ένας από τους δασκάλους έχει αποφασίσει να φύγει, αλλά δεν μπορεί ακόμη να το κάνει καθώς περιμένει να μειωθεί ο αριθμός των παιδιών στο χώρο (κρίσιμο τμήμα). Τι συμβαίνει στο σχήμα συγχρονισμού σας για τα νέα παιδιά που καταφτάνουν και επιχειρούν να μπουν στο χώρο;***

   Όταν ένας δάσκαλος επιχειρήσει να βγει ελέγχει αν το επιτρέπει η αναλογία παιδιών που περιμένουν να μπουν και υπολειπόμενων δασκάλων που είναι μέσα. Τα παιδιά που εισέρχονται ελέγχουν αν υπάρχουν αρκετοί δάσκαλοι ώστε να ισχύει η αναλογία για να μπουν αλλιώς περιμένουν. Αν μπει δάσκαλος ή βγει παιδί αποστέλλεται σήμα στα παιδιά που περιμένουν να μπουν και στους δάσκαλους που περιμένουν βγουν εκείνη τη στιγμή να συνεχίσουν. Ο πρώτος που θα πάρει το κλειδί ελέγχει αν το επιτρέπει η συνθήκη του (στο while) να συνεχίσει αλλιώς ξανακάνει wait και στέλνει το κλειδί σε άλλους που περιμένουν.

2. ***Υπάρχουν καταστάσεις συναγωνισμού (races) στον κώδικα του kgarten.c που επιχειρεί να επαληθεύσει την ορθότητα του σχήματος συγχρονισμού που υλοποιείτε; Αν όχι, εξηγήστε γιατί. Αν ναι, δώστε παράδειγμα μιας τέτοιας κατάστασης.***

   Καταστάσεις συναγωνισμού στον κώδικα που επιχειρεί να επαληθεύσει την ορθότητα του σχήματος συγχρονισμού (συνάρτηση verify()) υπάρχουν κάθε φορά που πάνω από ένα thread ζητούν το κλειδί (&thr->kg->mutex) για να μπουν σε κρίσιμο τμήμα. Παράδειγμα, όταν πάει να εκτελεστεί η verify() και ζητά το κλειδί, την ίδια ώρα που επιχειρεί ένα παιδί ή δάσκαλος να εισέλθει ή να εξέλθει. Οπότε ανταγωνίζονται, εκείνη την στιγμή, για το ποιος θα εκτελέσει πρώτος τον κώδικά του. Στην περίπτωση που δεν προλάβει πρώτη η verify() και εκτελεστεί κάποια είσοδος/έξοδος, ο έλεγχος που θα γίνει στην συνέχεια από τη verify(), δεν θα είναι αυτός που αρχικά είχε σκοπό να κάνει(θα έχουν μεταβληθεί τα vc και vt). Καταστάσεις συναγωνισμού υπάρχουν, επίσης, κάθε φορά που πάνω από ένα thread ζητούν το κλειδί στις συναρτήσεις εισόδου και εξόδου των παιδιών ή των δασκάλων και συναγωνίζονται για το ποιος θα μπορέσει να πάρει το κλειδί πρώτος και να εκτελέσει την αντίστοιχη είσοδο/έξοδο.