ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Λειτουργικά Συστήματα Υπολογιστών

3η εργαστηριακή αναφορά: Συγχρονισμός

Διδάσκοντες:

Ν. Κοζύρης
Γ. Γκούμας

oslaba84:

Ειρήνη Δόντη
ΑΜ 03119839

6ο εξάμηνο

Αθήνα 2022

Περιεχόμενα:

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Χρησιμοποιούμε το παρεχόμενο Makefile για να μεταγλωττίσουμε και να τρέξουμε το πρόγραμμα.
Παρατηρούμε ότι κάνοντας make, το αποτέλεσμα του simplesync που παράγεται διαφέρει από εκτέλεση σε εκτέλεση και είναι διάφορο του μηδενός.
Ωστόσο αυτό είναι λογικό καθώς έχουμε έναν επεξεργαστή που κάθε φορά εκτελείται για κάποιο από τα δύο νήματα.

```
oslaba84@os-node2:~/mysyncr/sync$ make
gcc -Wall -O2 -pthread -c -o pthread-test.o pthread-test.c
gcc -Wall -O2 -pthread -o pthread-test pthread-test.o
gcc -Wall -O2 -pthread -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c
gcc -Wall -O2 -pthread -o simplesync-mutex simplesync-mutex.o
gcc -Wall -O2 -pthread -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
gcc -Wall -O2 -pthread -c -o ex3_1_1.o ex3_1_1.c
ex3_1_1.c: In function ‘increase_fn’:
ex3_1_1.c:54:25: warning: value computed is not used [-Wunused-value]
                         __sync_fetch_and_add(&ip,1);
                         ^
ex3_1_1.c: In function ‘decrease_fn’:
ex3_1_1.c:79:25: warning: value computed is not used [-Wunused-value]
                         __sync_fetch_and_sub(&ip,1);
                         ^
gcc -Wall -O2 -pthread -o ex3_1_1 ex3_1_1.o
```

Ερωτήσεις:

1. Χρησιμοποιούμε την εντολή time(1) για να μετρήσουμε τον χρόνο εκτέλεσης των εκτελέσιμων. Τα αποτελέσματα που λαμβάνουμε είναι τα ακόλουθα:

```
oslaba84@os-node1:~/mysyncr/sync$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m26.513s
user    0m25.144s
sys     0m27.836s
```

```
oslaba84@os-node1:~/mysyncr/sync$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m0.126s
user    0m0.200s
sys     0m0.000s
```

Συγκρίνουμε τον χρόνο εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό με τον χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό. Παρατηρούμε ότι όταν έχουμε συγχρονισμό, οι χρόνοι εκτέλεσης είναι μεγαλύτεροι από τον χρόνο του αρχικού προγράμματος.
Στο αρχικό πρόγραμμα δεν μπορεί να ελεγχθεί ποιο από τα δύο νήματα θα χρησιμοποιήσει τους πόρους του συστήματος, καθώς έχουμε έλλειψη συγχρονισμού. Αντίθετα, στο πρόγραμμα με τον συγχρονισμό μόνο ένα νήμα κάθε φορά θα βρίσκεται στο κρίσιμο τμήμα, άρα μόνο αυτό θα χρησιμοποιεί τους πόρους.

2. Η πιο γρήγορη μέθοδος συγχρονισμού, είναι εκείνη με τη χρήση atomic operations. Η συγκεκριμένη υλοποίηση χρησιμοποιεί ειδικές εντολές (builtins) του επεξεργαστή, χωρίς να εκτελείται η κλήση συστήματος sleep() για κάποιο νήμα. Όταν χρησιμοποιούμε atomic operations, τότε υπάρχει ανάγκη ειδικών εντολών για την ατομική εκτέλεση των σύνθετων εντολών. Αντίθετα, η διαδικασία κλείδωμα-ξεκλείδωμα του mutex είναι πιο αργή, καθώς εκτελούνται επιπλέον οι κλήσεις sleep() και awake() για τα νήματα που πρέπει να εισέλθουν σε κάποιο κρίσιμο τμήμα.

3. Χρησιμοποιούμε την παρακάτω εντολή, ώστε να παράξουμε, σε ξεχωριστό αρχείο simplesync.s, τον ενδιάμεσο κώδικα Assembly για την περίπτωση που κάνουμε χρήση των Atomic Operations:

```
gcc -pthread -DSYNC_ATOMIC -S -g simplesync.c
```

Παρακάτω, απεικονίζεται τμήμα του κώδικα Assembly που προκύπτει για το Thread που έχει αναλάβει την αύξηση, καθώς η εντολή __sync_add_and_fetch(ip,1) μεταφράζεται ως εξής:

```
.L3:
        .loc 1 53 0
        lock addq       $1, -16(%rbp)
        .loc 1 49 0
        addl    $1, -4(%rbp)
```

Παρακάτω, απεικονίζεται τμήμα του κώδικα Assembly που προκύπτει για το Thread που έχει αναλάβει την μείωση., καθώς η εντολή __sync_sub_and_fetch(ip,1) μεταφράζεται ως εξής:

```
.L7:
        .loc 1 78 0
        lock subq       $1, -16(%rbp)
        .loc 1 74 0
        addl    $1, -4(%rbp)
```

Παρατηρούμε ότι οι ατομικές εντολές μεταφράζονται σε μία μόνο εντολή assembly.

4. Όπως στο προηγούμενο παράδειγμα, χρησιμοποιούμε την παρακάτω εντολή, ώστε να παράξουμε, σε ξεχωριστό αρχείο .s, τον ενδιάμεσο κώδικα Assembly για την περίπτωση λειτουργίας pthread_mutex_lock() και pthread_mutex_unlock() :

```
gcc -pthread -DSYNC_MUTEX -S -g simplesync.c
```

Παρακάτω, απεικονίζεται τμήμα του κώδικα Assembly που προκύπτει στην περίπτωση λειτουργίας pthread_mutex_lock() και pthread_mutex_unlock() αντίστοιχα:

```
.L3:
        .loc 1 56 0
        movl    $mutex1, %edi
        call    pthread_mutex_lock
```

```
.loc 1 59 0
movl    $mutex1, %edi
call    pthread_mutex_unlock
```

Ο κώδικας που χρησιμοποιήσαμε για το μέρος 1 είναι ο ακόλουθος:

```
/*
 * 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 mutex1 = PTHREAD_MUTEX_INITIALIZER; ///////////////////////

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);//////////////////////
                } else {
                        /* ... */
                        pthread_mutex_lock(&mutex1);
                        /* You cannot modify the following line */
                         ++(*ip);
                        pthread_mutex_unlock(&mutex1);//////////////////////
                        /* ... */
                }
        }
        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 {
                        pthread_mutex_lock(&mutex1);
                        /* You cannot modify the following line */
                        --(*ip);
                        pthread_mutex_unlock(&mutex1);//////////////////////
                }
        }
       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);

        return ok;
}
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

1. Οι σημαφόροι που χρειάζονται για το σχήμα συγχρονισμού είναι ίσοι σε πλήθος με εκείνο των νημάτων.

2. Χρησιμοποιούμε την εντολή cat /proc/cpuinfo για παρατηρούμε πως διαθέτουμε 4 υπολογιστικούς πυρήνες.

```
cpu cores       : 4
```

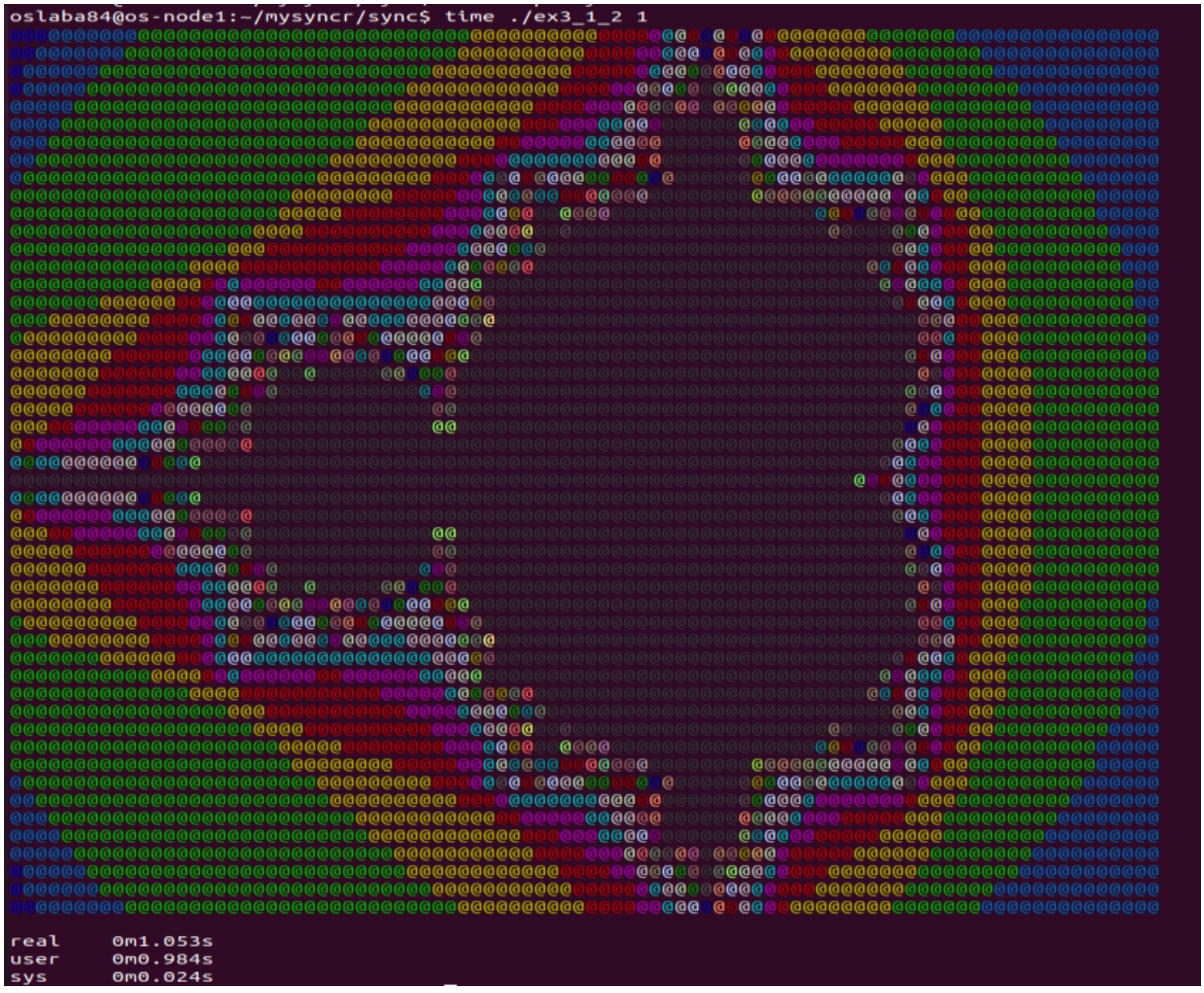
Χρησιμοποιούμε τις εντολές **time ./mandel 2** και **time ./ex3_1_2 2** για να υπολογίσουμε τον χρόνο που απαιτείται για την ολοκλήρωση του σειριακού και του παράλληλου προγράμματος με 2 νήματα υπολογισμού σε μηχάνημα δύο πυρήνων. Τα αντίστοιχα αποτελέσματα που λαμβάνουμε είναι τα ακόλουθα:

```
real    0m1.033s        real    0m0.527s
user    0m0.984s        user    0m0.992s
sys     0m0.024s        sys     0m0.020s
```

3. Το παραγόμενο παράλληλο πρόγραμμα εμφανίζει επιτάχυνση, καθώς γίνεται πρώτα ο υπολογισμός των γραμμών και έπειτα η εμφάνιση των στοιχείων τους. Το κρίσιμο τμήμα είναι η εκτύπωση του εκάστοτε στοιχείου της κάθε γραμμής. Η κάθε γραμμή που, υπολογιζόμενη από ένα νήμα, τυπώνεται και έπειτα παίρνει τον έλεγχο ένα άλλο νήμα για μία άλλη γραμμή, θα μείωνε την απόδοση, καθώς θα εκτελούνταν το πρόγραμμα σειριακά.

4. Πατώντας Ctrl+C πριν ολοκληρωθεί η διαδικασία εκτέλεσης του προγράμματος, το τερματικό εμφανίζει ό,τι έχουμε κάνει με χρώμα, καθώς δεν έχουν γίνει reset τα χρώματα. Επεκτείνουμε το mandel.c, ώστε να κάνουμε reset εντός του sigint_handler.

Ενδεικτικά, τρέχουμε τον κώδικα για 1 thread και λαμβάνουμε το κάτωθι αποτέλεσμα:



Ο κώδικας που χρησιμοποιήσαμε στο μέρος 2 είναι ο ακόλουθος:

```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 <semaphore.h>
#include <signal.h>
#include <pthread.h>
#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

/***************************
 * Compile-time parameters *
 ***************************/
#define perro_pthread(ret,msg)\
        do{errno = ret; perror(msg);} while(0)
/*
 * Output at the terminal is is x_chars wide by y_chars long
 */
int y_chars = 50; // number of lines
int x_chars = 90; // number of columns

/*
 * 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;

/*=========================================*/
void sigint_handler(int signum)
{
        reset_xterm_color(1); // reset all character attributes before leaving, to ensure the prompt is not drawn in a funny color, fd=1
        exit(1);
}

sem_t *mutex;


struct thread_info_struct
{
        pthread_t tid;
        int *color_val;
        int thrid;
        int thrcnt;
};

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 usage(char *argv0)
{
        fprintf(stderr, "Usage: %s thread_count array-size\n\n", argv0);
        exit(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); // takes (x,y) point on the complex plane ar
rot set
                if (val > 255) // check if color values are greater than 255 (the maximum color value)
                        val = 255;

                /* And store it in the color_val[] array */
                val = xterm_color(val); // function takes a color value as returned by Mandelbrot iterations and uses the
                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++) { // for every line
                /* Set the current color, then output the point */
                set_xterm_color(fd, color_val[i]); // set color of characters
                if (write(fd, &point, 1) != 1) {
                        perror("compute_and_output_mandel_line: write point");
                        exit(1);
                }
        }

        /* Now that the line is done, output a newline character */
        if (write(fd, &newline, 1) != 1) {
                perror("compute_and_output_mandel_line: write newline");
                exit(1);
        }
}

void *compute_and_output_mandel_line(void *arg)
{
        /*
         * A temporary array, used to hold color values for the line being drawn
         */
        struct thread_info_struct *th=arg;
        int i;
        for(i=th->thrid; i<y_chars; i+=th->thrcnt)
        {
                compute_mandel_line(i,th->color_val); // compute line i of output as an array of x-char color values
                sem_wait(&mutex[i % th->thrcnt]); // lock semaphore
                output_mandel_line(1, th->color_val); // output array of x-chars color values to 256 color xterm
                sem_post(&mutex[(i+1) % th->thrcnt]); // unlocks semaphore
        }
        return NULL;
}

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

        xstep = (xmax - xmin) / x_chars; // dimensions of the character
        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.
         */
        int i; int ret; int thrcnt;
        struct thread_info_struct *th;

        if(argc != 2){ usage(argv[0]);}
        if(safe_atol(argv[1], &thrcnt) < 0 || thrcnt <=0){
                fprintf(stderr, "%s is not valid for thread_count\n", argv[1]);
                exit(1);
        }

        struct sigaction sa; // a struct that controls a signal
        sigset_t sigset;
        sa.sa_handler = sigint_handler; // a pointer to function sigint_handler assigned to handle the signal
        sa.sa_flags = SA_RESTART; // collection of flag bits that affect behavior of signals - SA_RESTART:  function restarts and not fail until specified
        sigemptyset(&sigset); // manipulate signal sets - initialise signal mask to exclude all signals
        sa.sa_mask = sigset;  // signal mask definition - a signal set identifies a set of signals that are to be added to the signal mask
        if(sigaction(SIGINT,&sa,NULL) < 0){ // sigaction() - examines and changes the action associated with the specific signal
                perror("sigaction");
                exit(1);
        }
        th = safe_malloc(thrcnt *sizeof(*th));
        mutex = safe_malloc(thrcnt *sizeof(sem_t));
        for(i=0; i<thrcnt; i++){

                th[i].thrid = i;
                th[i].thrcnt = thrcnt;
                th[i].color_val = safe_malloc(x_chars * sizeof(int));
                if (i==0) {
                        sem_init(&mutex[i],0,1); // initialise unnamed semaphore at the address pointed to by mutex[] with value=1
                }
                else {
                        sem_init(&mutex[i],0,0); // initialise with value=0
                }


                ret = pthread_create(&th[i].tid, NULL, compute_and_output_mandel_line, &th[i]);
                if(ret){
                        exit(1);
                }
        }
        for(i=0; i<thrcnt; i++){
                ret = pthread_join(th[i].tid, NULL); // wait for thread th[i] to terminate
                if(ret){
```

```
                exit(1);
            }
        }
        for(i=0; i<thrcnt; i++){
            sem_destroy(&mutex[i]); // destroy unnamed semaphore at the address pointed to by mutex[i]
        }
        reset_xterm_color(1); // reset all characters attributes before leaving,fd=1
        return 0;
```

1.3 Επίλυση προβλήματος συγχρονισμού

Αν εκτελέσουμε τον κώδικα του kgarten.c παρατηρούμε ότι η έλλειψη συγχρονισμού ανάμεσα στα νήματα οδηγεί σε καταστροφικά αποτελέσματα.

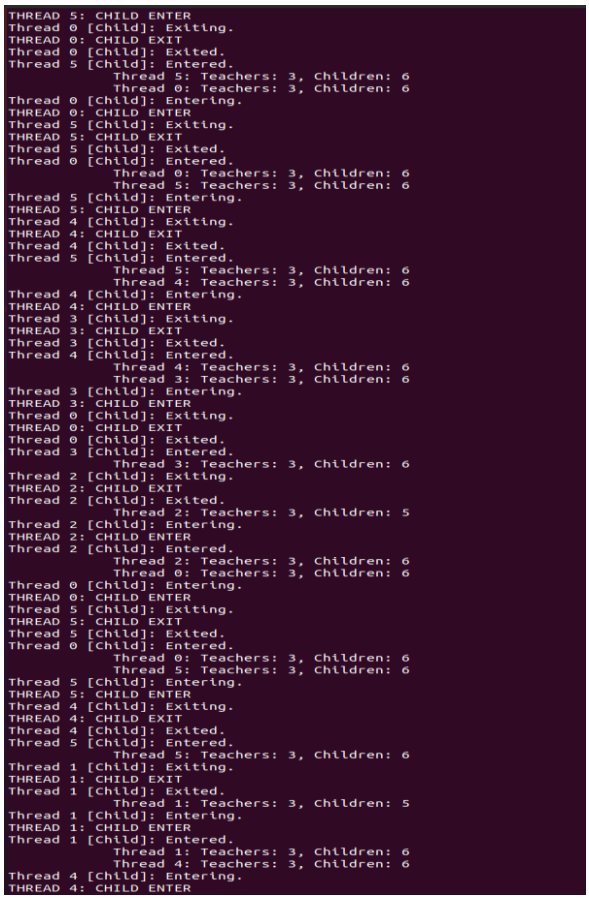
```
oslaba84@os-node1:~/mysyncr/sync$ ./ex3_1_3 10 7 2
Thread 1 of 10. START.
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 2 of 10. START.
Thread 9 of 10. START.
Thread 4 of 10. START.
Thread 4 [Child]: Entering.
THREAD 4: CHILD ENTER
Thread 7 of 10. START.
Thread 7 [Teacher]: Entering.
THREAD 7: TEACHER ENTER
Thread 9 [Teacher]: Entering.
Thread 5 of 10. START.
Thread 5 [Child]: Entering.
THREAD 9: TEACHER ENTER
Thread 9 [Teacher]: Entered.
            Thread 9: Teachers: 2, Children: 2
Thread 6 of 10. START.
Thread 6 [Child]: Entering.
Thread 1 [Child]: Entered.
            Thread 1: Teachers: 2, Children: 2
Thread 3 of 10. START.
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER
Thread 2 [Child]: Entering.
THREAD 2: CHILD ENTER
Thread 2 [Child]: Entered.
            Thread 2: Teachers: 2, Children: 4
THREAD 6: CHILD ENTER
Thread 6 [Child]: Entered.
            Thread 6: Teachers: 2, Children: 5
THREAD 5: CHILD ENTER
Thread 3 [Child]: Entered.
Thread 7 [Teacher]: Entered.
*** Thread 6: Oh no! Little Steve put his finger in the wall outlet and got ele
ctrocuted!
*** Why were there only 2 teachers for 5 children?!
Thread 0 of 10. START.
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
```

Ερωτήσεις:

1. Ο δάσκαλος που αποφασίζει να φύγει, θα καλέσει τη συνάρτηση teacher_exit() και θα περιμένει το lock από το σημαφόρο όταν εισέλθει κάποιος δάσκαλος στο δωμάτιο ή όταν τα παιδιά μειωθούν. Στην περίπτωση που ένα παιδί θέλει να εισέλθει στο δωμάτιο, θα καλέσει τη συνάρτηση child_enter() και θα περιμένει το lock από τον σημαφόρο όταν κάποιο παιδί εξέλθει ή όταν εισέλθει κάποιος δάσκαλος στο δωμάτιο. Το παιδί καταφέρνει να εισέλθει στο δωμάτιο αν μεταβληθούν οι συνθήκες που απαιτούνται. Έτσι, υπάρχει περίπτωση ο δάσκαλος να μην εξέλθει από το δωμάτιο ακόμα κι αν είχε δηλώσει πως θέλει να εξέλθει πριν εισέλθει το παιδί.

2. Υπάρχουν καταστάσεις συναγωνισμού στον κώδικα του kgarten.c οι οποίες τροποποιήθηκαν, ώστε να μην οδηγείται, το σχήμα συγχρονισμού σε αδιέξοδο. Στο πρόγραμμα kgarten.c ορίζεται η συνάρτηση verify(), η οποία ελέγχει αν ο επιτρεπόμενος αριθμός παιδιών και δασκάλων τηρείται. Στην περίπτωση που δεν έχουμε τη σωστή αναλογία, το πρόγραμμα σταματάει και εμφανίζεται μήνυμα λάθους. Πριν κληθεί η συνάρτηση γίνεται ένα lock και μόλις αυτή επιστρέψει γίνεται το αντίστοιχο unlock. Αν δε γινόταν αυτό, τότε θα υπήρχε κίνδυνος η verify να δώσει λάθος αποτέλεσμα στην περίπτωση που ένα νήμα την καλούσε ενώ παράλληλα ένα άλλο νήμα τροποποιούσε κάποιο από τα πεδία της.

Στιγμιότυπο από το αποτέλεσμα που προκύπτει όταν τρέχουμε τον κώδικα:



Ο κώδικας που χρησιμοποιήσαμε στο μέρος 3 είναι ο ακόλουθος:

```
/*
 * 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.
         */

        /* ... */

        /*
         * You may NOT modify anything in the structure below this
         * point.
         */
        int vt;
        int vc;
        int ratio;

        pthread_mutex_t mutex;
pthread_cond_t cond_child;    //
pthread_cond_t cond_teacher; //
};

/*
 * 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;
};
```

```c
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 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)
{
        if (!thr->is_child) {
                fprintf(stderr, "Internal error: %s called for a Teacher thread.\n",
                        __func__);
                exit(1);
        }

        fprintf(stderr, "THREAD %d: CHILD ENTER\n", thr->thrid);
        pthread_mutex_lock(&thr->kg->mutex);

        while(thr->kg->vt*thr->kg->ratio <= thr->kg->vc) //
        pthread_cond_wait(&thr->kg->cond_child, &thr->kg->mutex); // if not enough teahcers for a new child -> block the thread of the child who wants to enter

        ++(thr->kg->vc);

        pthread_mutex_unlock(&thr->kg->mutex);
}

void child_exit(struct thread_info_struct *thr)
{

        if (!thr->is_child) {
                fprintf(stderr, "Internal error: %s called for a Teacher thread.\n",
                        __func__);
                exit(1);
        }

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

        pthread_mutex_lock(&thr->kg->mutex);
        --(thr->kg->vc);
 pthread_cond_broadcast(&thr->kg->cond_child); // unblock all threads which are blocked on child's condition variable
if((thr->kg->vt-1)*thr->kg->ratio <= thr-> kg->vc) // (N-C-1)*R <= C if the condition doesn't permit change of a state then . . .
    pthread_cond_broadcast(&thr->kg->cond_teacher);  //unblock all threads which are blocked on teacher's condition variable

        pthread_mutex_unlock(&thr->kg->mutex);
}

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

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

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

 pthread_cond_broadcast(&thr->kg->cond_teacher); //unblock all threads which are blocked on teacher's condition variable

 pthread_cond_broadcast(&thr->kg->cond_child);   //unblock all threads which are blocked on childs's condition variable


        pthread_mutex_unlock(&thr->kg->mutex);
}

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

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

        pthread_mutex_lock(&thr->kg->mutex);
 while((thr->kg->vt-1)*thr->kg->ratio <= thr->kg->vc) // (N-C-1)*R <= C while the condition doesn't permit change of a state then . . .

 pthread_cond_wait(&thr->kg->cond_teacher,&thr->kg->mutex); // block the thread of the teacher who wants to exit
        --(thr->kg->vt);
        pthread_mutex_unlock(&thr->kg->mutex);
}

/*
 * Verify the state of the kindergarten.
 */

// Check if the permitted number of teachers and children is valid

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, "             Thread %d: Teachers: %d, Children: %d\n",
                thr->thrid, t, c);

        if (c > t * r) { // if not enough teachers -> accident
                bad_thing(thr->thrid, c, t);
                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;
        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);
                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)); */
                pthread_mutex_lock(&thr->kg->mutex);
                verify(thr);
                pthread_mutex_unlock(&thr->kg->mutex);

                usleep(rand_r(&thr->rseed) % 1000000); // stop currently the execution of the thread

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

                if (thr->is_child)
                        child_exit(thr);
                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);

                pthread_mutex_lock(&thr->kg->mutex);
                verify(thr);
                pthread_mutex_unlock(&thr->kg->mutex);
        }

        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;
        struct kgarten_struct *kg;

        /*
         * Parse the command line
         */
        if (argc != 4)
                usage(argv[0]);
        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 kindergarten and random number generator
         */
        srand(time(NULL));

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

        ret = pthread_mutex_init(&kg->mutex, NULL); // initialize the mutex with default attributes
        if (ret) {
                perror_pthread(ret, "pthread_mutex_init");
                exit(1);
        }

        /* ... */

        /*
         * Create threads
         */
        thr = safe_malloc(thrcnt * sizeof(*thr)); // save place in the memory in order to create the thread

        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);
                thr[i].rseed = rand();
```

```
                /* Spawn new thread */
                ret = pthread_create(&thr[i].tid, NULL, thread_start_fn, &thr[i]);
                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); // wait for thread thr[i] to terminate and don't save the returned value
                if (ret) {
                        perror_pthread(ret, "pthread_join");
                        exit(1);
                }
        }

        printf("OK.\n");

        return 0;
}
```

Makefile

```
#
# Makefile
#

CC = gcc

# CAUTION: Always use '-pthread' when compiling POSIX threads-based
# applications, instead of linking with "-lpthread" directly.
CFLAGS = -Wall -O2 -pthread
LIBS =

all: pthread-test simplesync-mutex simplesync-atomic kgarten mandel ex3_1_2 ex3_1_3

## Pthread test
pthread-test: pthread-test.o
	$(CC) $(CFLAGS) -o pthread-test pthread-test.o $(LIBS)

pthread-test.o: pthread-test.c
	$(CC) $(CFLAGS) -c -o pthread-test.o pthread-test.c

## Simple sync (two versions)
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

## Kindergarten
kgarten: kgarten.o
	$(CC) $(CFLAGS) -o kgarten kgarten.o $(LIBS)

kgarten.o: kgarten.c
	$(CC) $(CFLAGS) -c -o kgarten.o kgarten.c



## ex3_1_2
ex3_1_2: mandel-lib.o ex3_1_2.o
	$(CC) $(CFLAGS) -o ex3_1_2 mandel-lib.o ex3_1_2.o $(LIBS)

ex3_1_2.o: ex3_1_2.c
	$(CC) $(CFLAGS) -c -o ex3_1_2.o ex3_1_2.c $(LIBS)

##ex3_1_3
ex3_1_3: ex3_1_3.o
	$(CC) $(CFLAGS) -o ex3_1_3 ex3_1_3.o $(LIBS)

ex3_1_3.o: ex3_1_3.c
	$(CC) $(CFLAGS) -c -o ex3_1_3.o ex3_1_3.c $(LIBS)

## Mandel
mandel: mandel-lib.o mandel.o
	$(CC) $(CFLAGS) -o mandel mandel-lib.o mandel.o $(LIBS)

mandel-lib.o: mandel-lib.h mandel-lib.c
	$(CC) $(CFLAGS) -c -o mandel-lib.o mandel-lib.c $(LIBS)

mandel.o: mandel.c
	$(CC) $(CFLAGS) -c -o mandel.o mandel.c $(LIBS)

clean:
	rm -f *.s *.o pthread-test simplesync-{atomic,mutex} kgarten mandel
```