



Εθνικό Μετσόβιο Πολυτεχνείο

**Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών**

Λειτουργικά Συστήματα

3^η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Ημερομηνία Παράδοσης : 22/12/2017

Ονοματεπώνυμο	Αριθμός Μητρώου
----------------------	------------------------

Αλεξάκης Κωνσταντίνος	03114086
-----------------------	----------

Βασιλάκης Εμμανουήλ	03114167
---------------------	----------

Εξάμηνο	Ακαδημαϊκό Έτος
----------------	------------------------

7 ^ο	2017-2018
----------------	-----------

1 Συγχρονισμός σε υπάρχοντα κώδικα

1.1 Πηγαίος 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

/* 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;

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_fetch_and_add(ip, 1);
        } else {
            int ret;
            ret = pthread_mutex_lock(&mutex);
            //++(*ip);
        }
    }
}
```

```

        if (ret) {
            perror_pthread(ret, "mutex_lock");
            exit(1);
        }
        /* You cannot modify the following line */
        ++(*ip);
        ret = pthread_mutex_unlock(&mutex);
        if (ret) {
            perror_pthread(ret, "mutex_unlock");
            exit(1);
        }
    }
}
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_fetch_and_sub(ip, 1);
        } else {
            int ret;
            ret = pthread_mutex_lock(&mutex);
            if (ret) {
                perror_pthread(ret, "mutex_lock");
                exit(1);
            }
            /* You cannot modify the following line */
            --(*ip);
            ret = pthread_mutex_unlock(&mutex);
            if (ret) {
                perror_pthread(ret, "mutex_unlock");
                exit(1);
            }
        }
    }
    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 Έξοδος εκτέλεσης του προγράμματος

Αν τρέξουμε τα εκτελέσιμα *simplesync-atomic* και *simplesync-mutex* που παράγονται από το Makefile, θα λάβουμε ακριβώς την ίδια έξοδο:

```

About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

```

1.3 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;

Ο χρόνος εκτέλεσης του αρχικού προγράμματος είναι μικρότερος από τον χρόνο που απαιτείται για το συγχρονισμένο πρόγραμμα, είτε ο συγχρονισμός γίνεται με `mutex`, είτε με χρήση ατομικών εντολών, που είναι λογικό, αφού στην μία περίπτωση υπάρχει ένας επιπλέον περιορισμός που πρέπει να τηρηθεί και οι αντίστοιχοι έλεγχοι που γίνονται για να μην βρεθούμε σε κατάσταση μη συγχρονισμού.

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

Πιο γρήγορη είναι η χρήση ατομικών λειτουργιών, αφού δεν έχουμε εμπλοκή του Λειτουργικού Συστήματος, σε αντίθεση με την περίπτωση των `POSIX mutexes`, παρά μόνο οδηγία στον επεξεργαστή να μην διακόψει την εκτέλεση της εντολής. Επιπλέον, στην περίπτωση των `mutexes`, χρειάζεται να επικοινωνήσουμε με το ΛΣ για να κλειδώσουμε το `mutex`, μετά να κάνουμε την αύξηση ή την μείωση, και τέλος να επικοινωνήσουμε και πάλι με το ΛΣ και να ελευθερώσουμε το `mutex`, ενώ με την χρήση ατομικών λειτουργιών πραγματοποιείται η αύξηση ή η μείωση απλά χωρίς να διακοπεί η λειτουργία της.

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 πηγαιού 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`.

Η `__sync_fetch_and_add(ip, 1)` μεταφράζεται στις εντολές

```
movq -8(%rbp), %rax
lock addl $1, (%rax)
```

Η `__sync_fetch_and_sub(ip, 1)` μεταφράζεται στις εντολές

```
movq -8(%rbp), %rax
lock subl $1, (%rax)
```

Για να προκύψει το αρχείο με τον κώδικα assembly, εκτελέσαμε την εντολή **`gcc -g -S -DSYNC_ATOMIC simplesync.c`**

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.

Η `pthread_mutex_lock(&mutex)` μεταφράζεται στις εντολές

```
movl $mutex, %edi
call pthread_mutex_lock
movl %eax, -12(%rbp)
```

Η `pthread_mutex_unlock(&mutex)` μεταφράζεται στις εντολές

```
movl $mutex, %edi
call pthread_mutex_unlock
movl %eax, -12(%rbp)
```

Για να προκύψει το αρχείο με τον κώδικα assembly, εκτελέσαμε την εντολή **`gcc -g -S -DSYNC_MUTEX simplesync.c`**

2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

2.1 Πηγαίος 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 <errno.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>

#include "mandel-lib.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 MANDEL_MAX_ITERATION 100000

/* Signal handler for SIGINT */
void user_pressed_Ctrl_C (int signum)
{
    reset_xterm_color(1);
    if (signal (SIGINT, SIG_DFL) == SIG_ERR) {
        perror("SIGNAL_DEFAULT\n");
        exit(1);
    }
    if (raise (SIGINT)) {
        printf("Could not raise SIGINT");
        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;
}

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;
}

/*
 * A (distinct) instance of this structure
 * is passed to each thread
 */
struct thread_info_struct {
    int num;                //number of this thread
    int Number_Of_Threads;  //how many threads in total
    pthread_t tid;          /* POSIX thread id, as returned by the library */

    sem_t *mysem, *nextsem; /* Pointer to the semaphores I care about */
                           // mysem -> myturn
                           // nextsem -> next thread ready
};

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

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

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

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

```



```

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

    int n;
    int val;

    /* Find out the y value corresponding to this line */
    y = ymax - ystep * line;

    /* and iterate for all points on this line */
    for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {

        /* Compute the point's color value */
        val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
        if (val > 255)
            val = 255;

        /* And store it in the color_val[] array */
        val = xterm_color(val);
        color_val[n] = val;
    }
}

/*
 * This function outputs an array of x_char color values
 * to a 256-color xterm.
 */
void output_mandel_line(int fd, int color_val[])
{
    int i;

    char point = '@';
    char newline = '\n';

    for (i = 0; i < x_chars; i++) {
        /* Set the current color, then output the point */
        set_xterm_color(fd, color_val[i]);
        if (write(fd, &point, 1) != 1) {
            perror("compute_and_output_mandel_line: write point");
            exit(1);
        }
    }

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

```

```

}

void *thread_start_fn (void *arg)
{
    /* We know arg points to an instance of thread_info_struct */
    struct thread_info_struct *thr = arg;
    int i;

    for (i=thr->num; i<y_chars; i+=thr->Number_Of_Threads) {

        /*
         * A temporary array, used to hold color values for the line being drawn
         */
        int color_val[x_chars];
        compute_mandel_line(i, color_val);

        /* Wait for my turn */
        if (sem_wait(thr->mysem)) {
            perror("sem_wait");
            exit(1);
        }

        output_mandel_line(1, color_val);

        /* Next thread is ready */
        if (sem_post(thr->nextsem)) {
            perror("sem_post");
            exit(1);
        }
    }

    /* Thread is exiting */
    return NULL;
}

int main(int argc, char *argv[])
{
    int NTHREADS, i, ret;
    struct thread_info_struct *thr;

    /* Parse the command line */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s NTHREADS\n\n"
            "Exactly one argument required:\n"
            "  NTHREADS: The number of threads to create.\n",
            argv[0]);
        exit(1);
    }

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

    /* Done with parsing */

    xstep = (xmax - xmin) / x_chars;

```

```

ystep = (ymax - ymin) / y_chars;

sem_t sem[NTHREADS];          //create multiple semaphores & initialise
if (sem_init(&sem[0], 0, 1)) {
    perror("Semaphore not initialised correctly\n");
    exit(1);
}
for (i=1; i<NTHREADS; i++) {
    if (sem_init(&sem[i], 0, 0)) {
        perror("Semaphore not initialised correctly\n");
        exit(1);
    }
}

thr = safe_malloc(NTHREADS * sizeof (struct thread_info_struct));

if (signal (SIGINT, user_pressed_Ctrl_C) == SIG_ERR) {
    perror("SIGNAL_HANDLER\n");
    exit(1);
}

for (i=0; i<NTHREADS; i++) {
    /* Spawn new thread */
    thr[i].num = i;
    thr[i].Number_Of_Threads = NTHREADS;
    thr[i].mysem = &sem[i];
    thr[i].nextsem = &sem[(i+1) % NTHREADS];
    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<NTHREADS; i++) {
    ret = pthread_join(thr[i].tid, NULL);
    if (ret) {
        perror_pthread(ret, "pthread_join");
        exit(1);
    }
}

/* Destroy all semaphores and free */
for (i=0; i<NTHREADS; i++) {
    if (sem_destroy(&sem[i])) {
        perror("sem_destroy");
        exit(1);
    }
}
free(thr);
reset_xterm_color(1);
return 0;
}

```

2.2 Έξοδος εκτέλεσης του προγράμματος

Αν τρέξουμε το εκτελέσιμο *mandel* με είσοδο 2, θα λάβουμε την ακόλουθη έξοδο:

```
manolis@hp-255-g5:~/Desktop/OS/EX/2$ ./mandel 2
```

2.3 Ερωτήσεις

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

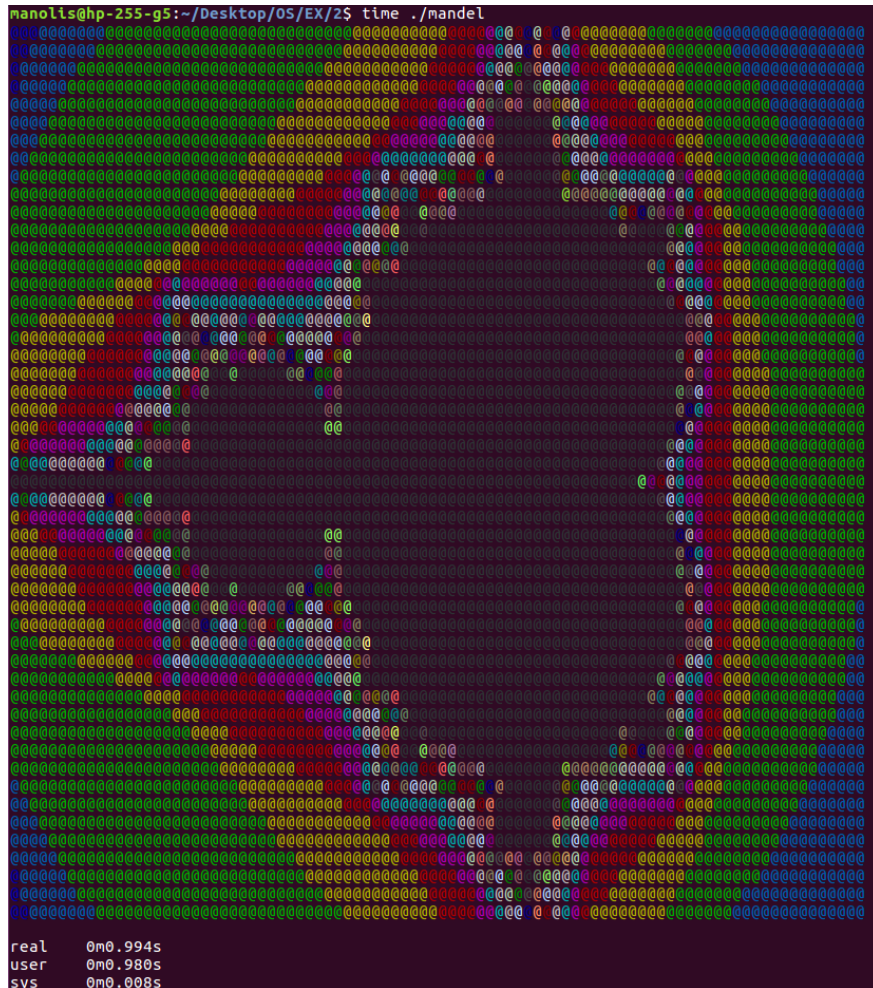
Χρειάζονται *NTHREADS* σημαφόροι.

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/cruinfo` για να δείτε πόσους υπολογιστικούς πυρήνες διαθέτει κάποιο μηχάνημα.

Παραθέτουμε τα αποτελέσματα στις παρακάτω φωτογραφίες, όπου πρώτα είναι η εικόνα που προκύπτει από την εκτέλεση του σειριακού προγράμματος, ενώ μετά από το παράλληλο.

```
nanolis@hp-255-g5:~/Desktop/05/EX/2$ time ./mandel
real    0m0.994s
user    0m0.980s
sys     0m0.008s
```



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;

Παρατηρούμε ότι το χρώμα των χαρακτήρων δεν είναι πλέον άσπρο. Για να επαναφέρουμε το χρώμα των γραμμάτων, υλοποιούμε έναν *signal_handler*, ο οποίος, αν μας έρθει σήμα *SIGINT*, το οποίο παράγεται από το πάτημα των *Ctrl-C*, αρχικά θα καλεί την *reset_terminal(1)*, έπειτα θα ορίζει ότι το *SIGINT* γίνεται *handle* με τον *default* τρόπο, το οποίο γίνεται με την εντολή *signal (SIGINT, DFL)*, και τέλος παράγουμε ένα νέο σήμα *SIGINT*, κάνοντας *raise (SIGINT)*.

3 Επίλυση προβλήματος συγχρονισμού

3.1 Πηγαίος 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_thread(ret, msg) \
    do { errno = ret; perror(msg); } while (0)

```

```

/* A virtual kindergarten */
struct kgarten_struct {
    pthread_cond_t cond;
    /*
    * You may NOT modify anything in the structure below this
    * point.
    */
    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;

```



```

};

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);
    }

    int ret;
    ret = pthread_mutex_lock(&thr->kg->mutex);
    if (ret) {
        perror_pthread(ret, "mutex_lock");
        exit(1);
    }

    while((thr->kg->vt)*(thr->kg->ratio) == thr->kg->vc) {           //oso o max arithmos paidion ==
                                                                    //arithmos paidion, kane wait
        ret = pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
        if (ret) {
            perror_pthread(ret, "cond_wait");
            exit(1);
        }
    }

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

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

    ret = pthread_mutex_unlock(&thr->kg->mutex);
    if (ret) {
        perror_pthread(ret, "mutex_unlock");
        exit(1);
    }
}

```

```

    }
}

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);

    int ret;
    ret = pthread_mutex_lock(&thr->kg->mutex);
    if (ret) {
        perror_pthread(ret, "mutex_lock");
        exit(1);
    }

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

    ret = pthread_cond_broadcast(&thr->kg->cond);
    if (ret) {
        perror_pthread(ret, "cond_broadcast");
        exit(1);
    }

    ret = pthread_mutex_unlock(&thr->kg->mutex);
    if (ret) {
        perror_pthread(ret, "mutex_unlock");
        exit(1);
    }
}

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);

    int ret;
    ret = pthread_mutex_lock(&thr->kg->mutex);
    if (ret) {
        perror_pthread(ret, "mutex_lock");
        exit(1);
    }

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

    ret = pthread_cond_broadcast(&thr->kg->cond);
    if (ret) {

```

```

        perror_pthread(ret, "cond_broadcast");
        exit(1);
    }

    ret = pthread_mutex_unlock(&thr->kg->mutex);
    if (ret) {
        perror_pthread(ret, "mutex_unlock");
        exit(1);
    }
}

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);
    }

    int ret;
    ret = pthread_mutex_lock(&thr->kg->mutex);
    if (ret) {
        perror_pthread(ret, "mutex_lock");
        exit(1);
    }

    while(thr->kg->vc > (thr->kg->vt - 1) * thr->kg->ratio) {    //oso paidia > max paidia gia -1 daskalous
        ret = pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
        if (ret) {
            perror_pthread(ret, "cond_wait");
            exit(1);
        }
    }

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

    --(thr->kg->vt);

    ret = pthread_mutex_unlock(&thr->kg->mutex);
    if (ret) {
        perror_pthread(ret, "mutex_unlock");
        exit(1);
    }
}

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

    if (c > t * r) {
        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);

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

    // Added
    ret = pthread_cond_init(&kg->cond, NULL);
    if (ret) {
        perror_pthread(ret, "pthread_cond_init");
        exit(1);
    }
}

```

```

/*
 * Create threads
 */
thr = safe_malloc(thrcnt * sizeof(*thr));

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

/* Destroy cond & mutex */
ret = pthread_cond_destroy(&kg->cond);
if (ret) {
    perror_pthread(ret, "pthread_cond_destroy");
    exit(1);
}
ret = pthread_mutex_destroy(&kg->mutex);
if (ret) {
    perror_pthread(ret, "pthread_mutex_destroy");
    exit(1);
}

printf("OK.\n");
return 0;
}

```

3.2 Εξόδος εκτέλεσης του προγράμματος

Αν τρέξουμε το εκτελέσιμο *kgarten*, με είσοδο $N=10$, $C=7$ και $R=2$, θα δούμε ότι το πρόγραμμα δεν τερματίζει ποτέ, οπότε παραθέτουμε το αρχικό τμήμα της εξόδου του:

```

manolis@hp-255-g5:~/Desktop/OS/EX/3$ ./kgarten 10 7 2
Thread 0 of 10. START.
Thread 0 [Child]: Entering.
Thread 1 of 10. START.
Thread 1 [Child]: Entering.
Thread 3 of 10. START.
Thread 4 of 10. START.
Thread 4 [Child]: Entering.
Thread 5 of 10. START.
Thread 5 [Child]: Entering.
Thread 3 [Child]: Entering.
Thread 2 of 10. START.
Thread 2 [Child]: Entering.
Thread 6 of 10. START.
Thread 6 [Child]: Entering.
Thread 8 of 10. START.
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
Thread 7 of 10. START.
Thread 7 [Teacher]: Entering.
THREAD 7: TEACHER ENTER
Thread 9 of 10. START.
Thread 9 [Teacher]: Entering.
THREAD 9: TEACHER ENTER
Thread 9 [Teacher]: Entered.
    Thread 9: Teachers: 3, Children: 0
Thread 7 [Teacher]: Entered.
    Thread 7: Teachers: 3, Children: 0
Thread 8 [Teacher]: Entered.
THREAD 1: CHILD ENTER
Thread 1 [Child]: Entered.
    Thread 1: Teachers: 3, Children: 1
THREAD 4: CHILD ENTER
Thread 4 [Child]: Entered.
THREAD 5: CHILD ENTER
Thread 5 [Child]: Entered.
    Thread 5: Teachers: 3, Children: 3
THREAD 3: CHILD ENTER
Thread 3 [Child]: Entered.
THREAD 2: CHILD ENTER
Thread 2 [Child]: Entered.
    Thread 2: Teachers: 3, Children: 5
THREAD 6: CHILD ENTER
Thread 6 [Child]: Entered.
    Thread 6: Teachers: 3, Children: 6
    Thread 8: Teachers: 3, Children: 6
    Thread 4: Teachers: 3, Children: 6
    Thread 3: Teachers: 3, Children: 6
Thread 8 [Teacher]: Exiting.
Thread 7 [Teacher]: Exiting.
Thread 3 [Child]: Exiting.
THREAD 3: CHILD EXIT
Thread 3 [Child]: Exited.
THREAD 0: CHILD ENTER
Thread 0 [Child]: Entered.
    Thread 0: Teachers: 3, Children: 6
Thread 4 [Child]: Exiting.
THREAD 4: CHILD EXIT
Thread 4 [Child]: Exited.
    Thread 3: Teachers: 3, Children: 5
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER

```


3.3 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 καταφθάνουν και επιχειρούν να μπουν στο χώρο;

Τα νέα παιδιά που καταφθάνουν, όσο υπάρχει χώρος θα μπαίνουν στο νηπιαγωγείο χωρίς να μπλοκάρουν, διαφορετικά θα περιμένουν να έρθει νέος δάσκαλος ή να φύγει κάποιο παιδί και να δημιουργηθεί χώρος. Στην περίπτωση αυτή, παρατηρούμε ότι έχουμε μία κατάσταση στην οποία περιμένει ένας δάσκαλος για να φύγει και κάποια παιδιά για να μπουν. Αν κάποιο παιδί αποχωρίσει, ή μπει κάποιος καινούριος δάσκαλος, τότε θα ξυπνήσουν όλα τα νήματα που περιμένουν, αλλά δεν γνωρίζουμε αν θα μπει κάποιο νέο παιδί ή θα αποχωρίσει ο δάσκαλος που θέλει να φύγει, με την προϋπόθεση ότι μπορεί να φύγει και δεν θα χαλάσει την αναλογία δασκάλων – παιδιών. Η πολιτική του χρονοδρομολογητή είναι αυτή η οποία επιλέγει την σειρά με την οποία θα εκτελεστούν τα νήματα αφού ξυπνήσουν, κλειδώνοντας αντίστοιχα το mutex, και έτσι λύνεται το πρόβλημα με τον πιο δίκαιο τρόπο, χωρίς δηλαδή να δοθεί προτεραιότητα στον δάσκαλο ή στα παιδιά.

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.

Ναι, στον κώδικα του `kgarten.c` υπάρχουν καταστάσεις συναγωνισμού, με ένα απλό παράδειγμα να είναι η περίπτωση που περιγράψαμε στο προηγούμενο ερώτημα, όπου ένα παιδί περιμένει για να μπει και ένας δάσκαλος περιμένει για να αποχωρίσει. Αν μπει κάποιος δάσκαλος ή αποχωρίσει ένα παιδί, θα γίνει broadcast και θα ξυπνήσουν όλα τα νήματα που περιμένουν, δηλαδή ο δάσκαλος που θέλει να αποχωρήσει και το παιδί που θέλει να μπει. Αφού ξυπνήσουν, τα νήματα θα συναγωνιστούν για το mutex, χωρίς να ξέρουμε αν τελικά θα συνεχίσει την λειτουργία του το νήμα-δάσκαλος ή το νήμα-παιδί. Το ποιο από τα νήματα θα καταφέρει τελικά να κλειδώσει πρώτο το mutex εξαρτάται από τον χρονοδρομολογητή και την πολιτική που ακολουθεί. Αφού το νήμα που εκτελείται ολοκληρώσει την λειτουργία του, δηλαδή είτε μπει στο νηπιαγωγείο είτε αποχωρήσει από αυτό, το άλλο νήμα που είχε ξυπνήσει

δεν μπορεί πλέον να εκτελέσει την λειτουργία του, επομένως θα ξανακάνει *pthread_cond_wait()*.