3η Εργασία

στα

Λειτουργικά Συστήματα

Ομάδα D12

των: Καραμπάση Αικατερίνη 03112517

Μανδηλαράς Νικηφόρος 03112

Σ.Η.Μ.Μ.Υ.
Ε.Μ.Π.

2015-2016

Άσκηση 1.1 *Συγχρονισμός σε υπάρχοντα κώδικα*

Στην άσκηση μας ζητείται να συγχρονίσουμε έναν δοθέντα κώδικα. Αυτό γίνεται διότι καταφέρνουμε να ελέγξουμε την τιμή μίας μεταβλητής η οποία αλλάζει(αυξάνει ή μειώνεται) κατά τις ίδιες φορές, χωρίς όμως το τελικό της αποτέλεσμα να είναι μηδενικό, όπως θα έπρεπε, αφού η αρχική της τιμή ήταν 0. Η αύξηση και η μείωση γίνεται με τη χρήση 2 νημάτων. Επομένως, αυτός είναι και ο λόγος που η τελική τιμή της μεταβλητής μας δεν είναι σωστή. Επεξηγηματικά, το ένα νήμα αυξάνει κατά 1 σε κάθε επανάληψη και το άλλο μειώνει κατά 1. Η παραπάνω διαδικασία επαναλαμβάνεται N φορές. Για να λύσουμε το πρόβλημά μας θα συγχρονίσουμε τα 2 νήματα. Αυτό μπορούμε να το πετύχουμε είτε με τα Atomic Operations είτε με κλειδώματα, δηλαδή mutexes. Τα πρώτα μας τα παρέχει ο compiler κι εξάγονται από τον προγραμματιστή μέσω builtins. Ο κώδικας που παράγαμε εν τέλει ήταν ο παρακάτω:

```
/*
 * simplesync.c
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

/*
 * POSIX thread functions do not return error numbers in errno,
 * but in the actual return value of the function call instead.
 * This macro helps with error reporting in this case.
 */
#define perror_pthread(ret, msg) do { errno = ret; perror(msg); } while (0)

#define N 10000000

/* Dots indicate lines where you are free to insert code at will */
/* ... */
#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) {
                                        /* ... */
                                        /* You can modify the following line */
                                        __sync_fetch_and_add(ip,1);
                                        /* ... */
                                } else {
                                        /* ... */
                                        /* You cannot modify the following line */
                                        pthread_mutex_lock(&mutex);
                                        ++(*ip);
                                        pthread_mutex_unlock(&mutex);
                                        /* ... */
                                }
                        }
                        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 {
                        /* ... */
                        /* You cannot modify the following line */
                        pthread_mutex_lock(&mutex);
                        --(*ip);
                        pthread_mutex_unlock(&mutex);
                        /* ... */
                }
        }
        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;
}
```

Για να παράξουμε τα εκτελέσιμά μας χρησιμοποιούμε το Makefile που μας δόθηκε και κάθε φορά δίνουμε ως όρισμα να μας παραχθεί ένα εκ των συγχρονισμένων αρχείων. Αυτό ο compiler το επιτυγχάνει αποκλείοντας μέσω ελέγχων το σημείο του κώδικα που δεν χρειάζεται. Οι atomic ops που χρησιμοποιούμε είναι οι __sync_fetch_and_add() και η __sync_fetch_and_sub(), για την πρόσθεση και την αφαίρεση αντίστοιχα.

Ερωτήσεις:

1) Ο χρόνος του συγχρονισμένου μας αρχείου είναι:

   Με τη χρήση Atomic-Ops:

   real 0m0.512s

   user 0m1.021s

   sys 0m0.000s

   Με τη χρήση mutexes:

   real 0m1.944s

   user 0m2.338s

sys 0m1.533s

Σε αντίθεση με το χρόνο εκτέλεσης που έδωσαν τα μη συγχρονισμένα προγράμματά μας:

Με τη χρήση Atomic Ops:

real 0m0.042s

user 0m0.082s

sys 0m0.000s

Με τη χρήση mutexes:

real 0m0.040s

user 0m0.069s

sys 0m0.004s

Εύκολα παρατηρούμε πως ο χρόνος των συγχρονισμένων αρχείων μας είναι αρκετά μεγαλύτερος αυτού των αρχικών εκτελέσιμων. Ο λόγος για τον οποίο συμβαίνει αυτό είναι πως στην περίπτωση που έχουμε συγχρονίσει τα αρχεία μας τα διαφορετικά threads μοιράζονται ένα κομμάτι κώδικα που στην προκειμένη περίπτωση είναι μία μεταβλητή. Έτσι για να εκτελέσει κάποιο thread το συγκεκριμένο κομμάτι κώδικα πρέπει να περιμένει το άλλο να το απελευθερώσει, δηλαδή να υπολογίσει την τιμή του. Εν αντιθέσει με την άλλη περίπτωση, κατά την οποία δεν περιμένει κανένα thread να ολοκληρώσει την εκτέλεσή του το άλλο και εκτελείται, πάραυτα.

2) Όπως μπορούμε εύκολα να παρατηρήσουμε η χρήση των atomic ops είναι εμφανώς πιο γρήγορη από τη χρήση POSIX mutexes. Αυτό συμβαίνει διότι οι atomic ops υλοποιούνται από το σύστημά μας στη βάση του, δηλαδή από τον compiler, και επομένως είναι έτσι γραμμένα που να εξασφαλίζουν μεγάλη ταχύτητα εκτέλεσης. Από την άλλη, τα mutexes προγραμματίζονται σε πιο υψηλού επιπέδου γλώσσα κι επομένως χρειάζονται πιο πολύ χρόνο για να διαχειριστούν την εναλλαγή των threads.

3) Δίνοντας στον compiler την εντολή gcc -g -S simplesync.c κι έχοντας πρώτα αποκλείσει το κομμάτι κώδικα που επιτρέπει την πρόσβαση στα κομμάτια που αφορούν τα mutexes λαμβάνουμε:

```
.loc 1 50 0
movq    -8(%rbp), %rax
```

```
lock addl       $1, (%rax)

.loc 1 76 0
movq            -8(%rbp), %rax
lock subl       $1, (%rax)
```

4) Εργαζόμαστε αντιστοίχως με πριν, οπότε έχουμε για το κομμάτι κώδικα από το lock μεχρι και την κλήση του unlock:

```
.loc 1 55 0
movl            $mutex, %edi
call            pthread_mutex_lock
.loc 1 56 0
movq            -8(%rbp), %rax
movl            (%rax), %eax
leal            1(%rax), %edx
movq            -8(%rbp), %rax
movl            %edx, (%rax)
.loc 1 57 0
movl            $mutex, %edi
call            pthread_mutex_unlock

.loc 1 81 0
movl            $mutex, %edi
call            pthread_mutex_lock
.loc 1 82 0
movq            -8(%rbp), %rax
movl            (%rax), %eax
leal            -1(%rax), %edx
movq            -8(%rbp), %rax
movl            %edx, (%rax)
.loc 1 83 0
movl            $mutex, %edi
call            pthread_mutex_unlock
```

1.2 *Παράλληλος υπολογισμός του συνόλου Mandelbrot*

Σε αυτήν την άσκηση μας ζητήθηκε να σχεδιάσουμε το σύνολο Mandelbrot με τη χρήση σημαφόρων. Για να εκτελέσουμε το πρόγραμμα θα χρειαστούμε συνολικά Ν σημαφόρους που θα ορίσει κατά την εκτέλεση ο χρήστης και οι οποίοι θα χαρακτηρίζονται από το όνομά τους, την τιμή τους, αλλά και το σύνολό τους Ν, πράγματα τα οποία ορίζονται μέσα στη δομή τους. Το πρόγραμμά μας έχει ως εξής:

```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

#include "pthread.h"
#include "mandel-lib.h"
#include "semaphore.h"

#define MANDEL_MAX_ITERATION 100000


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

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

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

```
//construct semaphores
struct thread_struct{

                        //POSIX thread id
                        pthread_t thid;
                        //application's thread id
                        sem_t semid;
                        int tid;
                        int count;
                        struct thread_struct *next;
};


//on previous lab we were told atoi was not safe and not to use it
//so we create a function to convert characters to numbers
int conv_num(char *c, int *num){

                        long l;
                        char *ch;

                        l = strtol(c, &ch, 10);
                        if(c!=ch && *ch=='\0'){
                                *num = l;
                                return 0;
                        }
                        else{
                                return -1;
                        }
}

//without using this function we got segmentation fault or bus error
int dif_malloc(size_t size){

                        void *p;
                        if((p = malloc(size)) == NULL){
                                fprintf(stderr, "Out of memory\n");
                                exit(1);
                        }

                        return p;
}
```

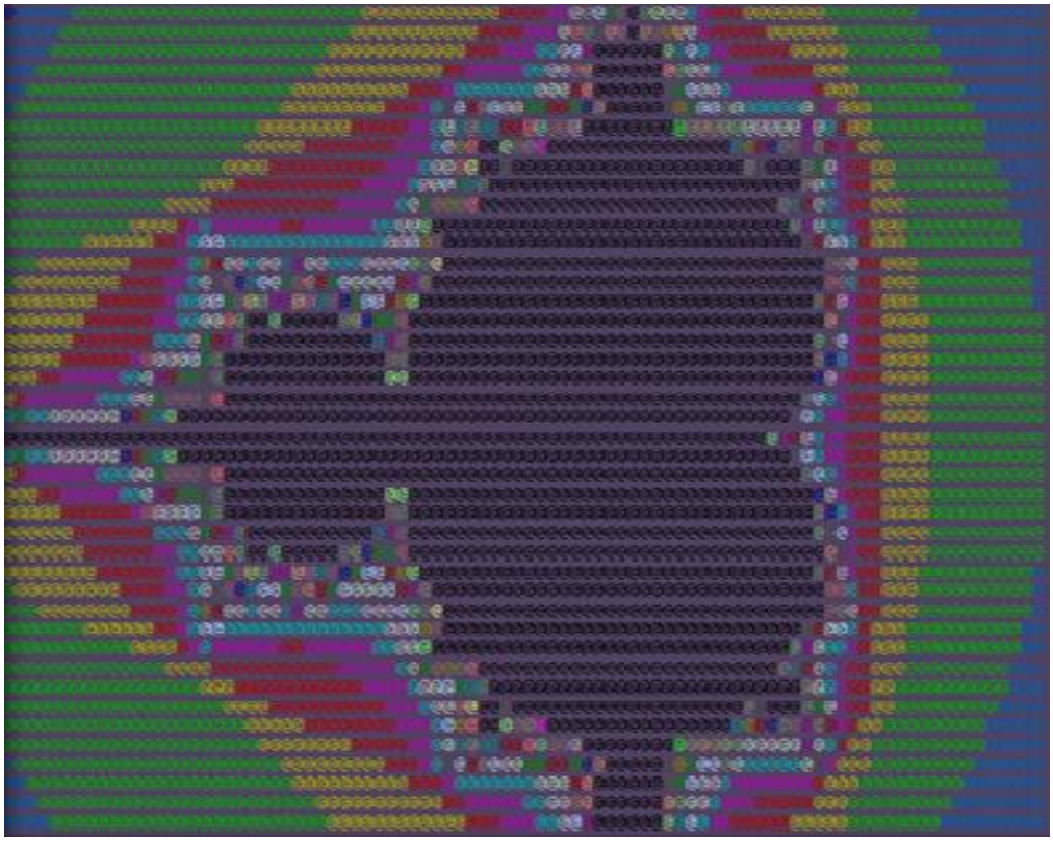
```
/*
 * This function computes a line of output
 * as an array of x_char color values.
 */
void compute_mandel_line(int line, int color_val[])
{
                        /*
                         * x and y traverse the complex plane.
                         */
                        double x, y;

                        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 *compute_and_output_mandel_line(void *arg)
{
                        /*
                         * A temporary array, used to hold color values for the line being drawn
                         */
                        int color_val[x_chars], i;
                        struct thread_struct *thr = arg;
                        for(i=thr->tid; i<y_chars; i+=thr->count){
                                compute_mandel_line(i, color_val);
                                sem_wait(&thr->semid);
                                output_mandel_line(1, color_val);
                                sem_post(&thr->next->semid);
                                reset_xterm_color(1);
                        }

                        return 0;
}


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

                        int i, counter, ret;
                        struct thread_struct *thr;
```

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

conv_num(argv[1], &counter);
thr = dif_malloc(counter *sizeof(*thr));
//we initilize the semaphores
sem_init(&thr[0].semid, 0, 1);
for(i=0;i<counter; i++){
        thr[i].tid = i;
        thr[i].count = counter;
        if(i!=(counter-1)){
                thr[i].next = &thr[i+1];
                sem_init(&thr[i+1].semid, 0, 0);
        }
        else{
                thr[i].next = &thr[0];
        }
}

/*
 * draw the Mandelbrot Set, one line at a time.
 * Output is sent to file descriptor '1', i.e., standard output.
 */
for (i = 0; i < counter; i++) {
        ret = pthread_create(&thr[i].thid, NULL,compute_and_output_mandel_line,
&thr[i]);
        if(ret){
                printf("pthread create\n");
                exit(1);
        }
}

for(i=0;i<counter; i++){
        ret = pthread_join(thr[i].thid, NULL);
        if(ret){
                printf("pthread join\n");
                exit(1);
        }
        sem_destroy(&thr[i].semid);
}
```

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

Η έξοδος που παίρνουμε στο τερματικό είναι η εξής:



Ερωτήσεις:

1) Όπως προειπώθηκε, στο πρόγραμμά μας για τη σωστή εκτέλεσή του θα χρειαστούμε Ν σημαφόρους τους οποίους θα τους ορίσει ο χρήστης κατά την εκτέλεση του προγράμματος. Τους σημαφόρους αυτούς, για τη σωστή εκτέλεση του προγράμματος άρα και για το σωστό αποτέλεσμα, τους συγχρονίζουμε. Ο συγχρονισμός αυτός θα γίνει ώστε να εκτυπώνεται η εικόνα μας κατά γραμμές με τη σειρά, δηλαδή η 1η γραμμή του 1ου σημαφόρου, η 1η γραμμή του 2ου σημαφόρου κ.ο.κ.

2) Μέσω της εντολής cat /proc/cpuinfo είδαμε πως ο υπολογιστής στον οποίο τρέχουμε το πρόγραμμά μας διαθέτει 4 επεξεργαστές. Επομένως, για αντιστοιχία στην εκφώνηση θα τρέξουμε την εντολή time ./mandel 4, οπότε έχουμε:

real 0m0.201s

user 0m0.613s

sys 0m0.024s

3) Το πρόγραμμά μας παρουσιάζει επιτάχυνση στην εκτέλεσή του όσο του δίνουμε ορίσματα μικρότερα ή ίσα του αριθμού των επεξεργαστών που διαθέτουμε. Για παράδειγμα αν δώσουμε ως όρισμα τον αριθμό 5 τότε αρχίζει και επιβραδύνει η εκτέλεση. Αυτό συμβαίνει διότι πρέπει να κάνει περισσότερες εναλλαγές μεταξύ του υπολογισμού και της εμφάνισης δηλαδή περισσότερες εναλλαγές για επικοινωνία λογισμικού και λειτουργικού. Το κρίσιμο τμήμα μας συγχρονίζεται μόνο στο κομμάτι της εμφάνισης του χρώματος. Αν συγχρονίζαμε και το κομμάτι του υπολογισμού τότε θα χρειαζόμασταν ακόμα πιο πολύ χρόνο, χωρίς αυτό να μας δινει κάποιο όφελος. Αυτό πρακτικά σημαίνει πως αν ο σημαφόρος μας είχε οριστεί πριν το τμήμα του υπολογισμού του χρώματος θα έκανε παραπάνω wait χωρίς λόγο.

4) Αυτό που θέλουμε να πετύχουμε είναι να διαχειρίζεται κάποιος το σήμα Ctrl+C και να επαναφέρει το terminal στην αρχική χρωματική του κατάσταση. Αυτό μπορούμε να το πετύχουμε με έναν signal handler, ο οποίος θα "πιάνει" το σήμα αυτό και θα μας επαναφέρει το terminal.

Συγκεκριμένα γράφουμε τη συνάρτηση:

```
void sighandler (int signo ) {
        if (signo == SIGINT ) {
            reset xterm color (1);
            exit (1);
        }
}
```

Επίσης στη συνάρτηση compute and output mandel line κάνουμε μια αλλαγή:

```
void *compute and output mandel line (void *arg ){
        /*
        * A temporary array , used to hold color values for the line being
```

```
                    */
                    int i , color val[x chars] ;
                    struct thread info struct *thr = arg ;
                    for(i=thr−>thrid ; i < ychars ; i+=thr−>thrcnt) {
                            compute mandel line ( i , colorval) ;
                            sem wait (& thr−>semid ) ;
                            signal(SIGINT , sighandler) ;
                            output mandel line (1 , color val ) ;
                            sempost(& thr−>next−>semid ) ;
                            reset xterm color (1);
                    }
                    return 0;
        }
```

1.3 *Επίλυση προβλήματος συγχρονισμού*

Η μορφή που έχει ο κώδικάς μας τελικά είναι η εξής:

```
#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 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;
   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;
};

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() % (sizeof(things)/sizeof(things[0]));
                        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);
        ++(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);
   if((thr->kg->vt - 1) * thr->kg->ratio <= thr->kg->vc)
      pthread_cond_broadcast(&thr->kg->cond_teacher);
        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);
    pthread_cond_broadcast(&thr->kg->cond_child);
                        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)
        pthread_cond_wait(&thr->kg->cond_teacher, &thr->kg->mutex);
                        --(thr->kg->vt);
                        pthread_mutex_unlock(&thr->kg->mutex);
}

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

        printf("OK.\n");

        return 0;
}
```

Ερωτήσεις:

1) Ο δάσκαλος θα περιμένει να φύγει επομένως θα μπει στο teacher_exit() και θα περιμένει το lock από το σημαφόρο όταν τα παιδιά μειωθούν, είτε όταν εισέλθει κάποιος άλλος δάσκαλος στο δωμάτιο. Αν ένα παιδί επιθυμήσει να εισέλθει στο χώρο τότε θα μπει στο child_enter() και θα περιμένει από το δικό του σημαφόρο το lock. Το παιδί θα εισέλθει στο χώρο είτε άμα κάποιο παιδί εξέλθει, είτε άμα εισέλθει κι άλλος δάσκαλος στο χώρο. Ωστόσο, υπάρχει πιθανότητα το παιδί να καταφέρει να εισέλθει στο χώρο αν υπάρξει μεταβολή στις συνθήκες που απαιτούνται και ο δάσκαλος να μη βγει ακόμα κι αν περιμένει πριν να θελήσει το παιδί να εισέλθει.

2) Η verify δε διασφαλίζει ότι ο έλεγχος γίνεται στην τρέχουσα κατάσταση. Δηλαδή, ελέγχει για τις μεταβλητές child και teacher αλλά τις τιμές αυτές δεν μπορεί να τις αλλάξει κάποιο άλλο thread εκείνη τη στιγμή. Έτσι, όταν κλήθηκε η verify, μπορεί οι τιμές με τις οποίες καλέστηκε να έχουν αλλάξει.

Προαιρετικές ερωτήσεις:

2. Παρατηρούμε πως ενώ με τη συνάρτηση rand() παράγονται ψευδο-τυχαίοι αριθμοί με βάση τη seed που έχουμε δώσει στην srand(), στη συγκεκριμένη περίπτωση ο αριθμός που τυπώνεται είναι ίδιος και για τις 10 επαναλήψεις του βρόχου χωρίς όρισμα, ενώ με όρισμα, όσο έχουμε δώσει. Το πρόβλημα έγκειται στον τρόπο με τον οποίο λειτουργεί η rand(). Για δεδομένο seed στην srand() κάθε φορά που θα εκτελείται η rand() για πρώτη φορά θα δίνει πάντα τον ίδιο αριθμό. Το seed έχει δοθεί στην srand() από τον πατέρα. Κάθε παιδί κληρονομεί τα χαρακτηριστικά του από τον πατέρα του, άρα και τον τρόπο με τον οποίο καλέσαμε την srand(). Όταν ένα παιδί καλέσει τη συνάρτηση rand() θα είναι η πρώτη φορά για το συγκεκριμένο παιδί. Συνεπώς, θα τυπώνεται ο ίδιος αριθμός για όλες τις διεργασίες.

3. Αν κάνουμε υλοποίηση με ένα σημαφόρο υπάρχει περίπτωση να θέλει να φύγει ένας καθητής και στην προσπάθειά του να μειώσει την τιμή του σημαφόρου να μπει κάποιο άλλο παιδί. Ως αποτέλεσμα, ο δάσκαλος δε θα μπορέσει εν τέλει να φύγει διότι ακόμα κι αν μειώσει την τιμή του σημαφόρου η διαδικασία μείωσης θα μείνει στη μέση και απλά θα προλάβει κάποιο παιδί να εισέλθει εκ νέου.