Λειτουργικά Συστήματα

3η αναφορά

Ονοματεπώνυμο	**Εμμανουηλίδης Εμμανουήλ**	**Λίτσος Ιωάννης**
Αριθμός Μητρώου	**03119435**	**03119135**
Ομάδα	**oslab61**	

Άσκηση 3:

ΣΥΓΧΡΟΝΙΣΜΟΣ

ΑΣΚΗΣΕΙΣ

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Χρησιμοποιώντας το παρεχόμενο Makefile μεταγλωττίζουμε και παράγουμε τα δύο διαφορετικά εκτελέσιμα αρχεία simplesync-atomic και simplesync-mutex από το ίδιο αρχείο πηγαίου κώδικα simplesync.c. Αυτό επιτυγχάνεται μέσω των εντολών -DSYNC_ATOMIC και –DSYNC_MUTEX που περιέχονται στο αρχείο Makefile. Πιο συγκεκριμένα, ανάλογα με το ποια από τις δύο εντολές χρησιμοποιούμε, κάνουμε define τη μία φορά το SYNC_ATOMIC και την άλλη το SYNC_MUTEX (παράγοντας το αντίστοιχο εκτελέσιμο). Έτσι, όπως μπορούμε να δούμε και στον κώδικα της simplesync.c :

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

καθορίζουμε ποια θα είναι η τιμή του USE_ATOMIC_OPS μέσω της οποίας καθορίζεται αν θα χρησιμοποιήσουμε mutexes ή atomic operations.

Εκτελώντας τα δύο αυτά προγράμματα (τα οποία αρχικά κάνουν την ίδια λειτουργία, εφόσον το σώμα της if είναι ίδιο με το σώμα της else και δεν τα έχουμε τροποποιήσει ακόμα) παίρνουμε το ακόλουθο output:

```
manos@manolo1742:~/Desktop/OS/exercises/3rd_exercise/sync$ ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = 3137001.
manos@manolo1742:~/Desktop/OS/exercises/3rd_exercise/sync$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -3000781.
```

Θα περίμενε κανείς ότι εφόσον αυξάνουμε και μειώνουμε ίσες φορές κατά 1 την τιμή του val(μέσω των increase_fn και decrease_fn) η τελική τιμή θα έπρεπε να είναι ίση με 0. Παρατηρούμε βέβαια ότι αυτό δε συμβαίνει. Αυτό συμβαίνει για τον εξής λόγο:

Αναθέτουμε την λειτουργία αύξησης της val σε ένα νήμα ενώ την λειτουργία μείωσης σε ένα άλλο νήμα. Ωστόσο, οι εντολές πρόσθεσης και αφαίρεσης μεταφράζονται η καθεμία σε

περισσότερες της μιας εντολής σε assembly. Έτσι, αν βρισκόμαστε στο κρίσιμο σημείο της πρόσθεσης ή της αφαίρεσης και συμβεί αλλαγή του νήματος που απασχολεί τον επεξεργαστή τότε θα προκύψει απρόβλεπτο αποτέλεσμα. Επομένως, αυτό συμβαίνει λόγω έλλειψης συγχρονισμού.

Επεκτείνουμε το κώδικα του αρχείου simplesync.c ώστε η εκτέλεση των δύο νημάτων στο εκτελέσιμο :

- simplesync-mutex να συγχρονίζεται με χρήση POSIX mutexes
- simplesync-atomic να συγχρονίζεται με χρήση ατομικών λειτουργιών του GCC

Ο κώδικας που προκύπτει είναι ο εξής:

```
/*
 * 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;

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_add_and_fetch(&ip,1);

        } else {
            pthread_mutex_lock(&mutex);
            /* You cannot modify the following line */
            ++(*ip);  /* critical section */
            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_sub_and_fetch(&ip,1);

        } else {
            pthread_mutex_lock(&mutex);
            /* You cannot modify the following line */
            --(*ip);  /* critical section */
            pthread_mutex_unlock(&mutex);
        }
    }
    fprintf(stderr, "Done decreasing variable.\n");

    return NULL;
```

```
}


int main (int argc, char *argv[])
{
    Int val, ret, ok;
    pthread_tt1, t2;
    pthread_mutex_init(&mutex, NULL);

    /*
     * 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;
}
```

Ο κώδικας είναι ίδιος με πριν με τις εξής αλλαγές:

Ορίζουμε ένα κλείδωμα mutex ως global μεταβλητή και την αρχικοποιούμε στην main() μέσω της pthread_mutex_init(&mutex,NULL) . Θέτοντας NULL αρχικοποιούμε το mutex στις προεπιλεγμένες ιδιότητές του. Επίσης, στη συνάρτηση *increase_fn, στο σώμα της if…else… εξετάζεται η μεταβλητή USE_ATOMIC_OPS. Αν είναι 1 ,δηλαδή αν έχουμε χρήση ατομικών λειτουργιών, καλείται η __sync_add_and_fetch(&ip,1) μέσω της οποίας αυξάνεται η μεταβλητή val κατά 1, ενώ αν είναι 0 , δηλαδή αν έχουμε χρήση mutexes, καλείται η pthread_mutex_lock(&mutex) προκειμένου να δεσμεύσουμε το κλείδωμα(mutex). Μετά, εκτελείται η εντολή του κρίσιμου τμήματος ++(*ip) και στη συνέχεια αποδεσμεύουμε το κλείδωμα με την εντολή pthread_mutex_unlock(&mutex).Αντίστοιχες εντολές έχουμε και στη συνάρτηση *decrease_fn.

Εκτελώντας τα δύο προγράμματα, λαμβάνουμε τις ακόλουθες εξόδους οι οποίες επιβεβαιώνουν την ορθή λειτουργία τους:

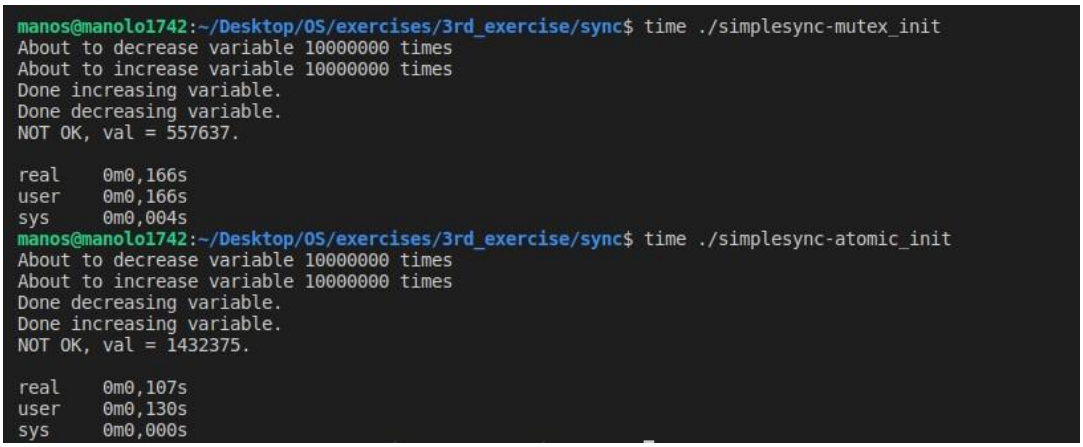
```
manos@manolo1742:~/Desktop/OS/exercises/3rd_exercise/sync$ ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

```
manos@manolo1742:~/Desktop/OS/exercises/3rd_exercise/sync$ ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

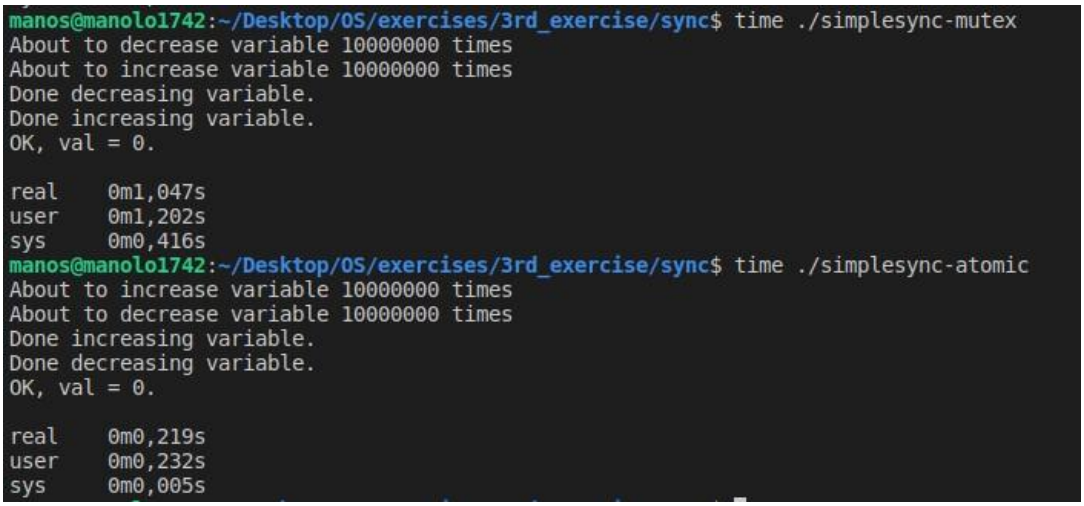
Ερωτήσεις

1.1 Χρησιμοποιούμε την εντολή timeγια να μετρήσουμε το χρόνο των εκτελέσιμων με και χωρίς συγχρονισμό(τα εκτελέσιμα αρχεία με κατάληξη _init είναι αυτά που προκύπτουν από το αρχικό αρχείο χωρίς συγχρονισμό) και παίρνουμε τα παρακάτω αποτελέσματα:

ΧΩΡΙΣ ΣΥΓΧΡΟΝΙΣΜΟ



ΜΕ ΣΥΓΧΡΟΝΙΣΜΟ



Όπως βλέπουμε οι χρόνοι εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό είναι μεγαλύτεροι από τους αντίστοιχους χρόνους των εκτελέσιμων χωρίς συγχρονισμό. Αυτό συμβαίνει διότι , όταν έχουμε συγχρονισμό , μόνο ένα νήμα μπορεί να βρίσκεται στο κρίσιμο τμήμα του κώδικα σε κάθε χρονική στιγμή , κάτι που δε συμβαίνει στην αντίθετη περίπτωση.

1.2 Όπως βλέπουμε, η υλοποίηση με χρήση ατομικών λειτουργιών είναι ταχύτερη απ' ότι η υλοποίηση με χρήση κλειδωμάτων mutexes και αυτό είναι λογικό. Οι ατομικές λειτουργίες είναι εγγυημένες να λειτουργούν ατομικά. Έτσι, αν εκτελούνται ταυτόχρονα (σε διαφορετικό πυρήνα), θα εκτελούνται ακολουθιακά με κάποια αυθαίρετη σειρά και δε θα έχουμε συνθήκες ανταγωνισμού. Επίσης, σε περίπτωση που κάποιο νήμα είναι απασχολημένο, επιτρέπουν στα υπόλοιπα νήματα να συνεχίσουν, εξασφαλίζοντας πως τουλάχιστον ένα νήμα εκτελεί χρήσιμο έργο. Αντίθετα, στην υλοποίηση με χρήση κλειδωμάτων mutexes, αν ένα νήμα κατέχει το κλείδωμα mutex, όλα τα υπόλοιπα νήματα μπλοκάρουν (sleep).Ένα αδρανές νήμα (sleep) περιμένει για το wakeup σήμα χωρίς να κάνει τίποτα. Αυτό έχει ως αποτέλεσμα να σπαταλούνται κύκλοι της CPU (κατά τη διάρκεια των suspend & wakeup) και να έχουμε περισσότερη καθυστέρηση. Ωστόσο, να σημειώσουμε πως οι ατομικές λειτουργίες, είναι χαμηλού επιπέδου, υλοποιούνται στο υλικό και δεν καλύπτουν ευρύ φάσμα εντολών, ενώ ακόμα δεν επιλύουν πάντα πλήρως τις συνθήκες συναγωνισμού. Έτσι πολλές φορές δεν είναι εύκολη (ή εφικτή) η χρήση τους και γι' αυτό τις περισσότερες φορές καταφεύγουμε σε άλλες μεθόδους συγχρονισμού.

1.3 Χρησιμοποιώντας την εντολή« gcc -S -g -DSYNC_ATOMIC simplesync.c » παράγουμε τον ενδιάμεσο κώδικα assembly του προγράμματος στη περίπτωση χρήσης ατομικών λειτουργιών. Το νήμα που εκτελεί την αύξηση χρησιμοποιεί την εντολή «__sync_add_and_fetch(&ip,1)» η οποία μεταφράζεται στις ακόλουθες εντολές του επεξεργαστή :

```
.L3:
    .loc 1 51 4
    lock addq   $1, -16(%rbp)
    .loc 1 47 22
    addl    $1, -20(%rbp)
```

Αντίστοιχα το νήμα που εκτελεί τη μείωση χρησιμοποιεί την εντολή «__sync_sub_and_fetch(&ip,1)» η οποία μεταφράζεται στις εντολές :

```
.L8:
    .loc 1 75 4
    lock subq   $1, -16(%rbp)
    .loc 1 71 22
    addl    $1, -20(%rbp)
```

1.4 Ακολουθώντας τα ίδια βήματα με πριν βρίσκουμε την μετάφραση σε assembly των POSSIX mutexes. Αυτή τη φορά χρησιμοποιούμε την εντολή « gcc -S -g -DSYNC_MUTEX simplesync.c ».Για το pthread_mutex_lock παράγεται ο ακόλουθος κώδικας:

```
.L3:
    .loc 1 54 4
    leaq    mutex(%rip), %rdi
    call    pthread_mutex_lock@PLT
```

Για το pthread_mutex_unlock παράγεται ακόλουθος κώδικας:

```
.loc 1 57 4
leaq    mutex(%rip), %rdi
call    pthread_mutex_unlock@PLT
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Σε αυτήν την άσκηση επεκτείνουμε το πρόγραμμα mandel.c έτσι ώστε ο υπολογισμός να κατανέμεται σε NTHREADS νήματα POSSIX με τον εξής τρόπο: για N νήματα το i-οστο νήμα αναλαμβάνει τις σειρές i, i+N , i+2N, … Ο ζητούμενος κώδικας είναι ο εξής:

```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 <errno.h>

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

#define MANDEL_MAX_ITERATION 100000

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

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

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

/*
```

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

/*
 * 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);
        }
    }
```

```c
    /* 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);
    }
}

/*---------------------------------------------------------------------*/

/*
 * 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)


void usage(char *argv0)
{
    fprintf(stderr, "Usage: %s wrong input!\n\n"
        "Exactly one argument required:\n"
        "    NTHREADS: The number of threads to create.\n",
        argv0);
    exit(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;
}

struct thread_info_struct {
    pthread_t tid;      /* POSIX thread id, as returned by the library */
    int thrcnt;         /* number of thread */
};

//declarations
sem_t *semaphores;                 // semaphores
int NTHREADS;                     // number of threads


void *compute_and_output_mandel_line(void *arg)
{
    struct thread_info_struct *thr = arg;

    int line = thr->thrcnt;
```

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

    /* thread i calculates lines: i, i + 1*NTHREADS, i + 2*NTHREADS...*/
    /* if line >  y_chars -> thread i has computed all of its lines */
    /* thread i releases semaphore for thread i+1 */
    for(i = line; i < y_chars; i+=NTHREADS) {
        compute_mandel_line(i,color_val);
        sem_wait(&semaphores[i % NTHREADS]);
        output_mandel_line(1, color_val);
        sem_post(&semaphores[(i+1) % NTHREADS]);
    }
    return NULL;
}


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;
}

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

    int i, ret;

    /*
     * Parse the command line
     */
    if (argc != 2)
        usage(argv[0]);
    if (safe_atoi(argv[1], &NTHREADS) < 0 || NTHREADS <= 0) {
        fprintf(stderr, "`%s' is not valid for\n", argv[1]);
        exit(1);
    }

    if(NTHREADS > y_chars) {
        printf("Too many threads! I can accept %i threads!\n", y_chars);
        exit(1);
    }

    struct thread_info_struct threads[NTHREADS];   // threads

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

```
    /* allocate memory for semaphores */
    semaphores = safe_malloc(NTHREADS * sizeof(sem_t));


    /* initialize semaphores */

    // #semaphores = NTHREADS

    /* the semaphore for the 1st thread (thread 0) , */
    /* needs to start with value = 1 */
    sem_init(&semaphores[0],0,1);

    // the rest of them start with value = 0
    for(i=1; i<NTHREADS; i++)
        sem_init(&semaphores[i],0,0);

    //create threads
    for(i=0; i < NTHREADS; i++) {

        threads[i].thrcnt = i;
        ret = pthread_create(&threads[i].tid, NULL, compute_and_output_mandel_line,
&threads[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(threads[i].tid, NULL);
        if (ret) {
            perror_pthread(ret, "pthread_join");
            exit(1);
        }
    }

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

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

Οι βασικές διαφορές εντοπίζονται στη main και στη συνάρτηση compute_and_output_mandel_line. Συγκεκριμένα στη main:

- Αρχικά ελέγχουμε για σωστό input και αποθηκεύουμε στη μεταβλητή NTHREADS τον αριθμό των νημάτων μέσω της safe_atoi, ενώ παράλληλα ελέγχουμε αν η μεταβλητή αυτή ξεπερνά την μεταβλητή y_chars δηλαδή τον αριθμό των γραμμών. Σε αυτή τη περίπτωση τυπώνεται το κατάλληλο μήνυμα και το πρόγραμμα κάνει exit.
- Στη συνέχεια μέσω της safe_malloc δεσμεύουμε χώρο για τους σημαφόρους και τους αρχικοποιούμε μέσω της sem_init όλους στη τιμή μηδέν εκτός από το πρώτο που ξεκινάει με τη τιμή 1.
- Έπειτα, μέσω της pthread_create δημιουργούμε τα νήματα και καλούμε την compute_and_output_mandel_line με παράμετρο την threads[i] που είναι ένας πίνακας τύπου thread_info struct ,δηλαδή περιέχει νήμα και τον αντίστοιχο αριθμό.
- Τέλος, περιμένουμε τα νήματα να τερματίσουν μέσω της pthread_join και καταστρέφουν τους σημαφόρους μέσω της sem_destroy.

Στην compute_and_output_mandel_line:

- Περνάμε ως παράμετρο μία μεταβλητή τύπου thread_info struct και το κύριο σώμα της αποτελείται από μία for όπου σε αυτήν αρχικά υπολογίζεται η γραμμή που πρόκειται να τυπωθεί . Έπειτα, καλείται η sem_wait για το σημαφόρο (i mod NTHREADS) . Αν η τιμή του σημαφόρου είναι μεγαλύτερη του 0 το νήμα συνεχίζει τυπώνοντας τη γραμμή μέσω της output_mandel_line και τέλος καλείται η sem_post για το σημαφόρο (i+1)mod(NTHREADS) προκειμένου να επιτραπεί στο επόμενο νήμα να εκτελέσει τη λειτουργία του

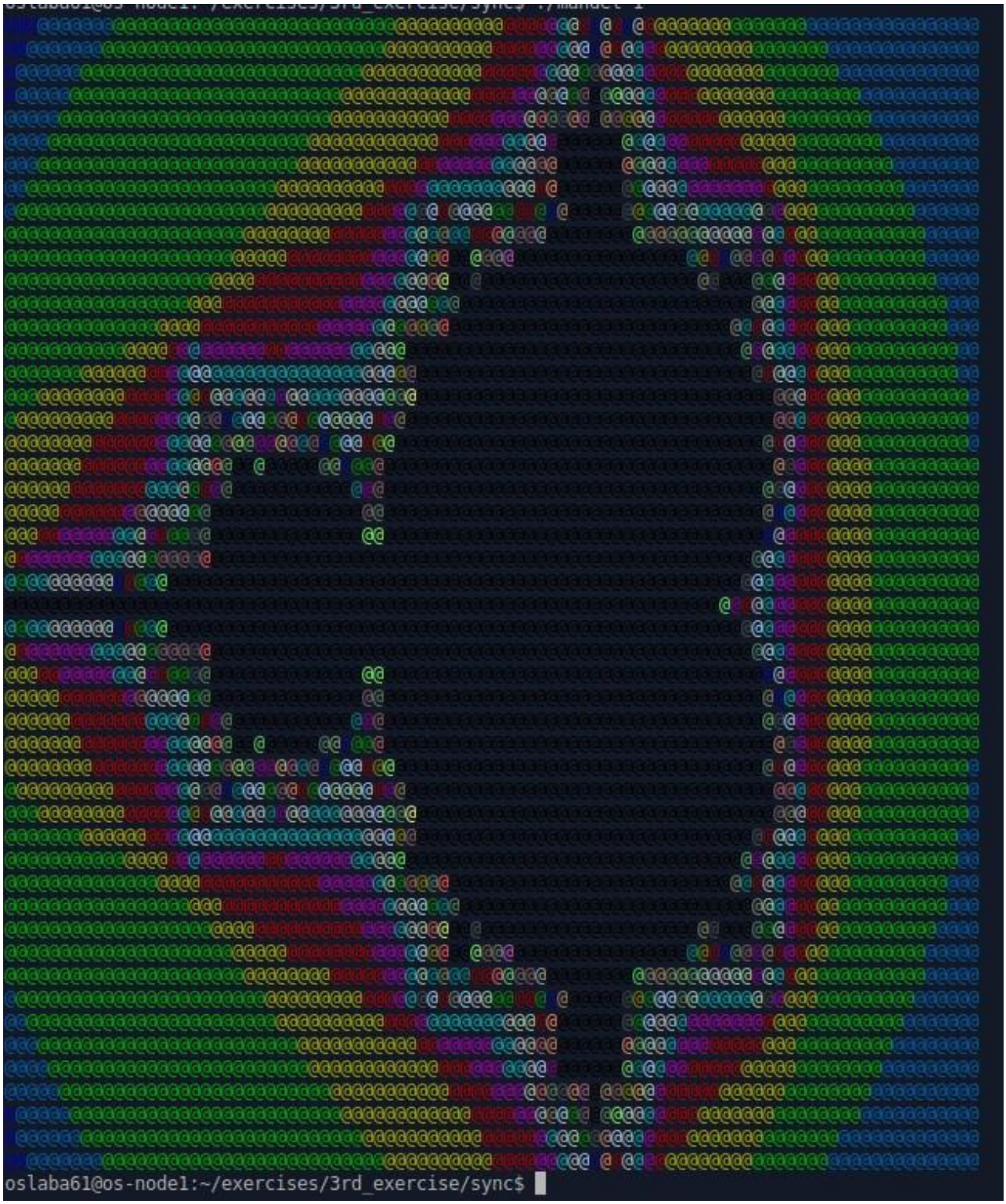
<u>Παρατήρηση</u>

Η for ξεκινά από τον αριθμό του νήματος και σε κάθε επανάληψη αυξάνεται κατά NTHREADS έως ότου η τιμή γίνει μεγαλύτερη από το συνολικό αριθμό γραμμών y_chars. Αυτό μας εξασφαλίζει ότι το i-οστο νήμα αναλαμβάνει τις σειρές i, i+N , i+2N,…. .

<u>Συμπέρασμα</u>

Με το συγχρονισμό μπορούμε να κάνουμε παράλληλα τον υπολογισμό n-γραμμών , αλλά ταυτόχρονα εξασφαλίζουμε ότι η διαδικασία της εκτύπωσης που αποτελεί και κρίσιμο τμήμα γίνεται με τη σωστή σειρά.

Το output που λαμβάνουμε όταν τρέχουμε το πρόγραμμα μας είναι το εξής:



ΕΡΩΤΗΣΕΙΣ

2.1 Εφόσον θέλουμε να γίνονται παράλληλα n υπολογισμοί γραμμών χρειαζόμαστε n νήματα και το κάθε νήμα χρειάζεται ένα σημαφόρο , οπότε συνολικά χρειαζόμαστε n σημαφόρους .

2.2 Τρέχοντας το πρόγραμμα με ένα νήμα βλέπουμε ότι η ολοκλήρωση του σειριακού προγράμματος απαιτεί τους εξής χρόνους:

Τρέχοντας το πρόγραμμα με δύο νήματα , η εκτέλεση του παράλληλου προγράμματος γίνεται στους εξής χρόνους:

Συγκρίνοντας τους παραπάνω χρόνους βλέπουμε ότι στη περίπτωση του παράλληλου προγράμματος ο πραγματικός χρόνος είναι ο μισός.

Σημείωση

Η εκτέλεση του προγράμματος έγινε στον orion όπου εκτελώντας την εντολή cat /proc/cpuinfo βλέπουμε ότι cpu_cores = 4.

2.3 Όπως φαίνεται και από το προηγούμενο ερώτημα το πρόγραμμα εμφανίζει επιτάχυνση. Αυτό οφείλεται στο γεγονός ότι το κρίσιμο τμήμα περιέχει μόνο την φάση εξόδου κάθε γραμμής που παράγεται. Έτσι, μπορούν τα n νήματα να εκτελούν παράλληλα τον υπολογισμό κάθε γραμμής εξοικονομώντας χρόνο και μόνο το κομμάτι της εξόδου εκτελείται σειριακά. Αν είχαμε συμπεριλάβει στο κρίσιμο τμήμα το κομμάτι του υπολογισμού των γραμμών τότε η λειτουργία του προγράμματος θα ήταν σειριακή, δηλαδή ένα νήμα θα υπολόγιζε και θα τύπωνε τη γραμμή του και μετά θα μπορούσε το επόμενο να ξεκινήσει τους δικούς του υπολογισμούς.

2.4 Εάν πατήσουμε Ctrl-C ενώ τρέχει ακόμα το πρόγραμμα δεν θα εκτελεστεί η εντολή reset_xterm_color και έτσι το χρώμα των γραμμάτων θα μείνει ίδιο με το χρώμα της τελευταίας γραμμής πριν διακοπεί το πρόγραμμα. Για να το αντιμετωπίσουμε αυτό φτιάχνουμε μία συνάρτηση που αναγνωρίζει τη διακοπή , τυπώνει κατάλληλο μήνυμα και

καλεί την reset_xterm_color για να επαναφέρει το σωστό χρώμα. Παρακάτω δίνεται η συνάρτηση που προστέθηκε στη mandel.c , η κλήση της μέσα στην main και μία ενδεικτική έξοδος που επιβεβαιώνει τη σωστή λειτουργία του προγράμματος.

```c
/* function to detect signal (ex. Ctrl C) and reset color */
void my_handler(sig_t s) {
    printf("Caught signal %d\n", s);
    reset_xterm_color(1);
    exit(1);
}
```

```c
signal (SIGINT,my_handler);
```

1.3 Επίλυση Προβλήματος Συγχρονισμού

Μελετώντας το κώδικα kgarten.c βλέπουμε ότι αυτός οδηγεί σε καταστροφικά αποτελέσματα το οποίο οφείλεται στην έλλειψη συγχρονισμού ανάμεσα στα νήματα διότι σε κάθε είσοδο παιδιού ή έξοδο δασκάλου δεν ελέγχεται εάν οι δάσκαλοι επαρκούν για να επιβλέψουν τα παιδιά.

Σε αυτή την άσκηση, λοιπόν, επεκτείνουμε το κώδικα του kgarten.c εφαρμόζοντας ένα σχήμα συγχρονισμού έτσι ώστε να μην οδηγούμαστε σε καταστροφικά αποτελέσματα .Πιο συγκεκριμένα, κάθε φορά που θέλει ένας δάσκαλος να φύγει ή ένα παιδί να μπει στο χώρο ελέγχουμε αν η ενέργεια αυτή παραβιάζει ή όχι την απαραίτητη συνθήκη :

(αριθμός δασκάλων)*(πλήθος παιδιών ανά δάσκαλο) ≥ πλήθος παιδιών

Σε περίπτωση που δεν παραβιάζεται η παραπάνω συνθήκη η ενέργεια πραγματοποιείται , διαφορετικά ο δάσκαλος ή το παιδί πρέπει να περιμένει.

Για το σκοπό αυτό έχουμε επεκτείνει τη δομή kgarten struct με μία μεταβλητή συνθήκης τύπου pthread_cond_t και έχουμε συμπληρώσει το σώμα των συναρτήσεων teacher_exit() , teacher_enter(), child_exit() και child_enter() ως εξής:

Child enter ():

Όταν ένα παιδί πάει να εισέλθει αφού δεσμευτεί το κλείδωμα μέσω της pthread_mutex_lock ελέγχουμε αν η είσοδος του παιδιού παραβιάζει την αναγκαία συνθήκη και αν συμβαίνει αυτό μπλοκάρουμε αυτή την ενέργεια μέσω της pthread_cond_wait η οποία ελευθερώνει το κλείδωμα mutex και μπλοκάρει το συγκεκριμένο νήμα. Διαφορετικά το παιδί εισέρχεται κανονικά και η ροή εκτέλεσης συνεχίζει.

Child exit ():

Όταν ένα παιδί πάει να φύγει ελέγχουμε αφού γίνει η έξοδος του αν ικανοποιείται η συνθήκη και σε περίπτωση που ικανοποιείται καλείται η pthread_cond_broadcast η οποία ξεμπλοκάρει όλα τα νήματα που έχουν μπλοκαριστεί μέσω της μεταβλητής συνθήκης που χρησιμοποιήθηκε στην pthread_cond_wait.

Teacher exit ():

Οι αλλαγές που κάνουμε εδώ είναι αντίστοιχες με αυτές που κάνουμε στην child_enter() διότι όταν πάει να φύγει ένας δάσκαλος θέλουμε να ελέγξουμε εάν οι εναπομείναντες δάσκαλοι είναι αρκετοί για να επιβλέψουν όλα τα παιδιά.

Teacher enter ():

Ομοίως και εδώ οι αλλαγές είναι αντίστοιχες με αυτές που κάναμε στην child_exit().

Τέλος, στη main έχουμε φροντίσει να αρχικοποιήσουμε τη μεταβλητή συνθήκης μέσω της pthread_cond_init , όπως και να την καταστρέψουμε μέσω της pthread_cond_destroy .

Ο κώδικας του προγράμματος είναι ο εξής:

```
/*
 * 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)
```

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

    /*
     * Here you may define any mutexes / condition variables / other variables
     * you may need.
     */

    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);
    }

    pthread_mutex_lock(&thr->kg->mutex);
    while( (thr->kg->vt * thr->kg->ratio < thr->kg->vc+1) || (thr->kg->vt == 0))  {
        /* while teachers * R < children+1 (OR there is no teacher) -> wait  */
        pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
    }
    fprintf(stderr, "THREAD %d: CHILD ENTER\n", thr->thrid);
    ++(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);
    }

    pthread_mutex_lock(&thr->kg->mutex);
    fprintf(stderr, "THREAD %d: CHILD EXIT\n", thr->thrid);
    --(thr->kg->vc);
    if(thr->kg->vt * thr->kg->ratio > thr->kg->vc) {
        pthread_cond_broadcast(&thr->kg->cond);
    }
    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);
    }

    pthread_mutex_lock(&thr->kg->mutex);
    fprintf(stderr, "THREAD %d: TEACHER ENTER\n", thr->thrid);
    ++(thr->kg->vt);
    if(thr->kg->vt * thr->kg->ratio >= thr->kg->vc) {
        pthread_cond_broadcast(&thr->kg->cond);
    }
    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);
    }

    pthread_mutex_lock(&thr->kg->mutex);
    while( ( (thr->kg->vt-1) * thr->kg->ratio < thr->kg->vc) || ( (thr->kg->vt-1 == 0) &&
(thr->kg->vc != 0) ) )  {
```

```
        /* while (teachers-1) * R < children (OR there is no teacher)  -> wait */
        pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
    }
    fprintf(stderr, "THREAD %d: TEACHER EXIT\n", thr->thrid);
    --(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, ret2, 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);
    }

    ret2 = pthread_cond_init(&kg->cond, NULL);
    if (ret2) {
        perror_pthread(ret2, "pthread_mutex_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 and destroy cond
     */
    for (i = 0; i < thrcnt; i++) {
        ret = pthread_join(thr[i].tid, NULL);
        ret2 = pthread_cond_destroy(&thr->kg->cond);
        if (ret) {
            perror_pthread(ret, "pthread_join");
            exit(1);
        }
        if (ret2) {
            perror_pthread(ret2, "pthread_cond_destroy");
            exit(1);
        }
    }

    printf("OK.\n");

    return 0;
}
```

Τρέχοντας το κώδικα βλέπουμε ότι το πρόγραμμα δεν τερματίζει ποτέ, δηλαδή δεν οδηγούμαστε ποτέ σε καταστροφική κατάσταση επιβεβαιώνοντας ότι ο συγχρονισμός λειτουργεί με τον επιθυμητό τρόπο.

Ερωτήσεις

3.1-3.2 Όταν ένας δάσκαλος αποφασίσει να φύγει αλλά δεν του επιτραπεί επειδή η έξοδος του θα παραβιάζει την απαραίτητη συνθήκη τότε αυτός θα αναγκαστεί να περιμένει να μειωθεί ο αριθμός των παιδιών στο χώρο. Ταυτόχρονα όμως μπορεί να επιτραπεί (εφόσον ισχύει η συνθήκη) σε νέα παιδιά να εισέλθουν στο χώρο. Γενικά είναι πιθανό να μπαίνουν και να βγαίνουν παιδιά με τέτοιο τρόπο ώστε η έξοδος του δασκάλου να παραβιάζει πάντα την συνθήκη και έτσι να μην μπορεί να φύγει ποτέ. Επίσης, υπάρχουν και καταστάσεις συναγωνισμού (races) .Για παράδειγμα, μπορεί να θέλουν να μπουν ορισμένα παιδιά στο χώρο αλλά να μην τους επιτραπεί λόγω ανεπάρκειας των δασκάλων με αποτέλεσμα αυτά τα παιδιά να μπλοκάρουν. Εάν μπει κάποιος δάσκαλος τότε τα παιδιά θα ξεμπλοκάρουν και θα συναγωνιστούν μεταξύ τους και θα μπουν μόνο όσα «προλάβουν» . Αντίστοιχες καταστάσεις συναγωνισμού παρουσιάζονται και όταν κάποιοι δάσκαλοι προσπαθούν ταυτόχρονα να φύγουν από το χώρο.

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 simplesync-
mutex_init simplesync-atomic_init mandel_init kgarten_init

## 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

## Simple sync initial (two versions)
simplesync-mutex_init: simplesync-mutex_init.o
    $(CC) $(CFLAGS) -o simplesync-mutex_init simplesync-mutex_init.o $(LIBS)

simplesync-atomic_init: simplesync-atomic_init.o
    $(CC) $(CFLAGS) -o simplesync-atomic_init simplesync-atomic_init.o $(LIBS)

simplesync-mutex_init.o: simplesync_init.c
```

```
	$(CC) $(CFLAGS) -DSYNC_MUTEX -c -o simplesync-mutex_init.o simplesync_init.c

simplesync-atomic_init.o: simplesync_init.c
	$(CC) $(CFLAGS) -DSYNC_ATOMIC -c -o simplesync-atomic_init.o simplesync_init.c

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

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

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

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


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

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

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
```