



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Λειτουργικά Συστήματα

7ο εξάμηνο, Ακαδημαϊκή περίοδος 2017-2018

Άσκηση 3: Συγχρονισμός



Team: oslaba27	
Ονοματεπώνυμο	Αριθμός Μητρώου
Κερασιώτης Ιωάννης	03114951
Ραφτόπουλος Ευάγγελος	03114743

Άσκηση 1.1

Πηγαίος Κώδικας:

simplesync.c

```

/*****
/*                                     Operating Systems                                     */
/*                                     Third Lab Exercise                                   */
/*                                     */
/* Exercise 1.1 Simplesync                                           */
/*                                     */
/* Team: oslaba27                                                    */
/*                                     */
/* Full Name: Kerasiotis Ioannis, Student ID: 03114951              */
/* Full Name: Raftopoulos Evangelos, Student ID: 03114743          */
/*                                     */
*****/

/*
 * 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_thread(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 cnt_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) {

            __sync_add_and_fetch(ip, 1);

        } else {

            pthread_mutex_lock(&cnt_mutex);
            ++(*ip);
            pthread_mutex_unlock(&cnt_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) {

        __sync_add_and_fetch(ip, -1);

    } else {

        pthread_mutex_lock(&cnt_mutex);
        --(*ip);
        pthread_mutex_unlock(&cnt_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_thread(ret, "pthread_join");
ret = pthread_join(t2, NULL);
if (ret)
    perror_thread(ret, "pthread_join");

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

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

return ok;
}

```

Έξοδος εκτέλεσης προγράμματος:

Η έξοδος του προγράμματος και στα δύο εκτελέσιμα προγράμματα είναι η παρακάτω:

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

Ερωτήσεις:

1. Χρησιμοποιήστε την εντολή `time(1)` για να μετρήσετε το χρόνο εκτέλεσης των εκτελέσιμων. Πώς συγκρίνεται ο χρόνος εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό, σε σχέση με το χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό; Γιατί;

Χρησιμοποιώντας την εντολή `time(1)` στα δύο εκτελέσιμα προγράμματα λαμβάνονται τα παρακάτω αποτελέσματα:

\$ time ./simplesync-atomic

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

real 0m0.413s
user 0m0.816s
sys 0m0.000s

\$ time ./simplesync-mutex

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

real 0m3.728s
user 0m3.796s
sys 0m2.904s

Εν αντιθέση το εκτελέσιμο χωρίς συγχρονισμό παίρνουμε σαν χρόνο εκτέλεσης:

real 0m0.038s
user 0m0.072s
sys 0m0.000s

Η εκτέλεση του εκτελέσιμο προγράμματος χωρίς συγχρονισμό είναι πιο γρήγορο καθώς δεν απαιτείται κάποιος περιορισμός στην εκτέλεση των δύο threads.

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

Η μέθοδος με χρήση ατομικών λειτουργιών είναι αισθητά γρηγορότερη και μπορεί να βγει το συμπέρασμα χωρίς κιάλας να γίνει χρήση της εντολής `time(1)`. Με την χρήση της εντολής γίνεται αντιληπτό ότι είναι γρηγορότερο κατά 3 sec. Αυτό οφείλεται στο ότι υλοποιείται με `atomic operations`, οι οποίες μεταφράζονται σε μια εντολή `assembly`, ενώ η υλοποίηση με `mutexes` αποτελεί μια πιο `high-level` προσέγγιση, η οποία ενσωματώνει την έννοια των `atomic operations` για να υλοποιηθεί.

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 πηγαίου κώδικα (π.χ., `".loc 1 63 0"`), οι οποίες μπορεί να σας διευκολύνουν. Δείτε την έξοδο της εντολής `make` για τον τρόπο μεταγλώττισης του `simplesync.c`.

Παρακάτω παρατέθεται ο κώδικας του `__sync_fetch_and_add` μεταφρασμένος σε `assembly`:

.L3:

```
.loc 1 50 0
movq  -16(%rbp), %rax
lock addl  $1, (%rax)
.loc 1 47 0
addl  $1, -4(%rbp)
```

Μετά τον κώδικα διαχείρισης στοίβας (για την δημιουργία του `stack frame`) για την κλήση της συνάρτησης, έχουμε την κύρια λειτουργία της `atomic` λειτουργίας, που υλοποιείται με την εντολή `lock addl`.

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.

Οι αντίστοιχες εντολές σε assembly είναι οι παρακάτω:

.L3:

```
.loc 1 54 0
movl  $cnt_mutex, %edi
call  pthread_mutex_lock
.loc 1 55 0
movq  -16(%rbp), %rax
movl  (%rax), %eax
leal  1(%rax), %edx
movq  -16(%rbp), %rax
movl  %edx, (%rax)
.loc 1 56 0
movl  $cnt_mutex, %edi
call  pthread_mutex_unlock
.loc 1 47 0
addl  $1, -4(%rbp)
```


Άσκηση 1.2

Πηγαίος Κώδικας:

mandel.c

```

/*****
/*                                     Operating Systems                               */
/*                                     Third Lab Exercise                             */
/*                                     */
/* Exercise 1.2 Mandel                                                         */
/*                                     */
/* Team: oslaba27                                                             */
/*                                     */
/* Full Name: Kerasiotis Ioannis, Student ID: 03114951                       */
/* Full Name: Raftopoulos Evangelos, Student ID: 03114743                   */
/*                                     */
*****/

/*
 * mandel.c
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

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

```

/*
 * Define a type of a struct which contain:
 *
 * a. A thread type ID (tid) for each thread which is created
 * b. An integer that resembles current line
 * c. A semaphore for current and next line
 */
typedef struct{
    pthread_t tid;
    int line;
    sem_t mutex;
}newThread_t;

/*
 * A pointer to newThread_t variable
 * for creation of new threads.
 * It is going to be allocated at main function
 * as NTHREAD array.
 */
newThread_t *thread;

/*
 * 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;

/*
 * The number of Threads created for parallel computation
 */
int NTHREADS;

```

```

/*
 * 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)
{
    int line = *(int *)arg;
    int index;

    /*
     * A temporary array, used to hold color values for the line being drawn
     */

    int color_val[x_chars];

    for(index=line; index<y_chars; index+=NTHREADS){

        /*
         * Compute mandel line
         */

        compute_mandel_line(index, color_val);

        /*
         * Wait the current line to be printed
         * and send sign to next semaphore
         */

        sem_wait(&thread[(index % NTHREADS)].mutex);
        output_mandel_line(STDOUT_FILENO, color_val); // file descriptor for stdout = 1 else
I could use STDOUT_FILENO processor symbol
        sem_post(&thread[((index % NTHREADS) + 1) % NTHREADS].mutex);
    }

    return NULL;
}

/*
 * This function provides user the safety for reset all character

```

```
* attributes before leaving,to ensure the prompt is  
* not drawn in a fancy color.  
*/
```

```
void unexpectedSignal(int sign){
```

```
    signal(sign, SIG_IGN);  
    reset_xterm_color(1);  
    exit(1);
```

```
}
```

```
int main(void)
```

```
{
```

```
    int line;
```

```
    /*  
    * For question 4, we should modify the program  
    * in order to terminate normally, if user send  
    * a signal interrupt from keyboard ( Ctrl + C )  
    */
```

```
    signal(SIGINT, unexpectedSignal);
```

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

```
    /*  
    * draw the Mandelbrot Set, one line at a time.  
    * Output is sent to file descriptor '1', i.e., standard output.  
    */
```

```
    /*  
    * Input of Number of threads  
    */
```

```
    printf("Enter Number of Threads: ");  
    scanf("%d", &NTHREADS);
```

```
    if(NTHREADS < 1 || NTHREADS > y_chars){  
        printf("Input is not valid\n");  
        return 0;  
    }
```

```
    /*  
    * Allocation of Threads and malloc check
```

```

*/

thread = (newThread_t *)malloc(NTHREADS * sizeof(newThread_t));
if(thread == NULL){
    perror("");
    exit(1);
}

/*
 * Initialization of semaphores.
 * The initialization of first mutex semaphore is set as not asleep,
 * contrastingly the others semaphores that they must set to wait.
 */

if((sem_init(&thread[0].mutex, 0, 1)) == -1){ // The sem_init function returns 0 on success and
-1 on error
    perror("");
    exit(1);
}

for(line=1; line<NTHREADS; line++){
    if((sem_init(&thread[line].mutex, 0, 0)) == -1){ // The sem_init function returns 0 on
success and -1 on error
        perror("");
        exit(1);
    }
}

/*
 * Creation of NTHREADS threads passing current line as argument to our
 * starting function, compute_and_output_mandel_line.
 */

for(line=0; line<NTHREADS; line++){

    thread[line].line = line;
    if((pthread_create(&thread[line].tid, NULL, compute_and_output_mandel_line,
&thread[line].line)) != 0){ //The pthread_create function return 0 on success
        perror("");
        exit(1);
    }
}

/*
 * The following loop makes sure that the terminating of threads will
 * not be after the done.
 */

```

```

    for(line=0; line<NTHREADS; line++){
        if((pthread_join(thread[line].tid, NULL)) != 0){ // As pthread_create. pthread_join
function return 0 on success
            perror("");
            exit(1);
        }
    }

    /*
     * Destroy the semaphores
     */

    for (line = 0; line < NTHREADS; line++) {
        sem_destroy(&thread[line].mutex);
    }

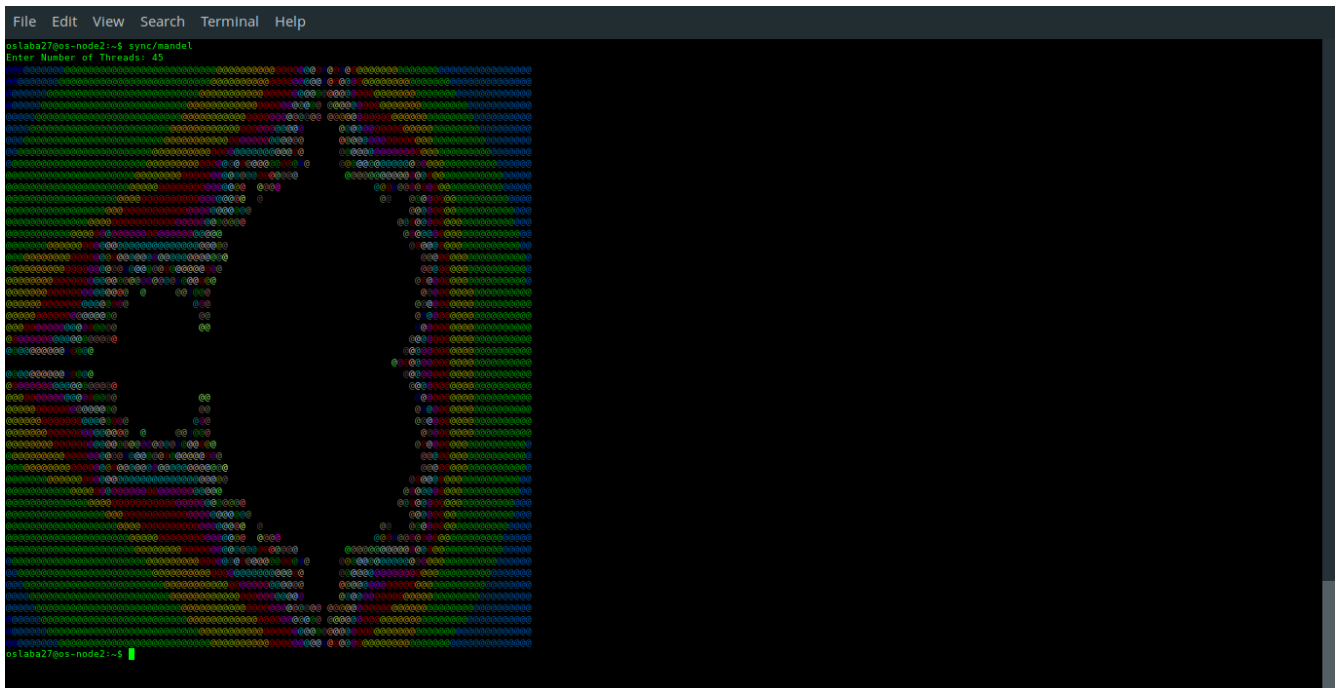
    free(thread);

    reset_xterm_color(1);
    return 0;
}

```

Έξοδος εκτέλεσης προγράμματος:

Η έξοδος του προγράμματος από το εκτελέσιμο πρόγραμμα παρατίθεται στην παρακάτω εικόνα:



```
File Edit View Search Terminal Help
ylaba27@os-node2:~$ sync/mandel
Enter Number of Threads: 45
[Output: A large grid of characters, including letters, numbers, and symbols, arranged in a pattern that suggests a complex data structure or a visualization.]
ylaba27@os-node2:~$
```

που είναι και το επιθυμητό αποτέλεσμα.

Ερωτήσεις:

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

Για την υλοποίηση χρησιμοποιούνται N σημαφόροι, ένα σημαφόρο για το κάθε thread. Για την ακρίβεια, ο σημαφόρος i ενεργοποιείται με την λήξη της εργασιάς του i-1 νήματος και σηματοδοτεί την έναρξη του N+1 νήματος. Πιο αναλυτικά το σχήμα συγχρονισμού του προγράμματος λειτουργεί σαν κυκλικός δακτύλιος, με τον κάθε νήμα να παίρνει την άδεια προς την εκτέλεση από το προηγούμενό του.

2. Πόσος χρόνος απαιτείται για την ολοκλήρωση του σειριακού και του παράλληλου προγράμματος με δύο νήματα υπολογισμού; Χρησιμοποιήστε την εντολή `time(1)` για να χρονομετρήσετε την εκτέλεση ενός προγράμματος, π.χ., `time sleep 2`. Για να έχει νόημα η μέτρηση, δοκιμάστε σε ένα μηχάνημα που διαθέτει επεξεργαστή δύο πυρήνων. Χρησιμοποιήστε την εντολή `cat /proc/cpuinfo` για να δείτε πόσους υπολογιστικούς πυρήνες διαθέτει κάποιο μηχάνημα.

Οι μετρήσεις είναι οι παρακάτω:

Σειριακός υπολογισμός		Παράλληλος με 4 threads	
real	0m0,548s	real	0m0,185s
user	0m0,534s	user	0m0,607s
sys	0m0,014s	sys	0m0,020s

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

Παρατηρώντας τις παραπάνω μετρήσεις το παράλληλο πρόγραμμα εμφανίζει επιτάχυνση δεδομένου, ότι η φάση υπολογισμού γίνεται παράλληλα και ο συγχρονισμός επιτυγχάνεται κατά την διάρκεια του τυπώματος, ώστε να γίνει με την σωστή σειρά. Σε άλλη περίπτωση δεν θα είχαμε την επιτάχυνση καθώς το κάθε νήμα θα περίμενε το προηγούμενο για να υπολογιστεί, με αποτέλεσμα το πρόγραμμα να εκφυλίζεται στο αντίστοιχο σειριακό.

4. Τι συμβαίνει στο τερματικό αν πατήσετε `Ctrl-C` ενώ το πρόγραμμα εκτελείται; σε τι κατάσταση αφήνεται, όσον αφορά το χρώμα των γραμμών; 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;

Με το πάτημα `Ctrl+C`, το πρόγραμμα τερματίζει καθώς του στέλνεται σήμα `SIGINT`, με αποτέλεσμα το χρώμα των γραμμών του τερματικού αραμένει σε αυτό που είχε επιλεγεί για τύπωμα ακριβώς πριν τερματιστεί. Δημιουργώντας τον δικό μας signal handler που με το πάτημα `Ctrl+C`, επαναφέρει το χρώμα του τερματικού πριν τερμαστεί.

Οι εντολές που χρησιμοποιούνται είναι οι παρακάτω:

```
signal(SIGINT, unexpectedSignal);
```

όπου unexpectedSignal είναι μια void συνάρτηση η οποία:

```
void unexpectedSignal(int sign){  
  
    signal(sign, SIG_IGN);  
    reset_xterm_color(1);  
    exit(1);  
}
```

Άσκηση 1.3

Πηγαίος Κώδικας:

kgarden.c

```

/*****
/*                                     Operating Systems                               */
/*                                     Third Lab Exercise                             */
/*                                     */
/* Exercise 1.3 Kgarden                                                           */
/*                                     */
/* Team: oslaba27                                                                */
/*                                     */
/* Full Name: Kerasiotis Ioannis, Student ID: 03114951                         */
/* Full Name: Raftopoulos Evangelos, Student ID: 03114743                     */
/*                                     */
*****/

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

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

```
#include <semaphore.h>
```

```
/*
```

```
 * POSIX thread functions do not return error numbers in errno,
```

```
 * but in the actual return value of the function call instead.
```

```
 * This macro helps with error reporting in this case.
```

```
*/
```

```
#define perror_pthread(ret, msg) \
    do { errno = ret; perror(msg); } while (0)
```

```
/* A virtual kindergarten */
```

```
struct kgarten_struct {
```

```
    /*
```

```
     * Here you may define any mutexes / condition variables / other variables
```

```
     * you may need.
```

```
    */
```

```
    /* a. an integer variable that resemble the space
```

```
     * b. a semaphore
```

```
     * c. a condVar */
```

```
    int space;
```

```
    sem_t door;
```

```
    pthread_cond_t condVar;
```

```
    /*
```

```
     * You may NOT modify or use anything in the structure below this
```

```
     * point. They are only meant to be used by the framework code,
```

```
     * for verification.
```

```
    */
```

```
    int vt;
```

```
    int vc;
```

```
    int ratio;
```

```
    pthread_mutex_t mutex;
```

```
};
```

```
/*
```

```
 * A (distinct) instance of this structure
```

```
 * is passed to each thread
```

```
*/
```

```
struct thread_info_struct {
```

```
    pthread_t tid; /* POSIX thread id, as returned by the library */
```

```
    struct kgarten_struct *kg;
```

```
    int is_child; /* Nonzero if this thread simulates children, zero otherwise */
```

```
    int thrid; /* Application-defined thread id */
```

```
    int thrcnt;
```

```

        unsigned int rseed;
    };

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

    l = strtol(s, &endp, 10);
    if (s != endp && *endp == '\0') {
        *val = l;
        return 0;
    } else
        return -1;
}

void *safe_malloc(size_t size)
{
    void *p;

    if ((p = malloc(size)) == NULL) {
        fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
            size);
        exit(1);
    }

    return p;
}

void usage(char *argv0)
{
    fprintf(stderr, "Usage: %s thread_count child_threads c_t_ratio\n\n"
        "Exactly two argument required:\n"
        "  thread_count: Total number of threads to create.\n"
        "  child_threads: The number of threads simulating children.\n"
        "  c_t_ratio: The allowed ratio of children to teachers.\n\n",
        argv0);
    exit(1);
}

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

```

```

char *things[] = {
    "Little %s put %s finger in the wall outlet and got electrocuted!",
    "Little %s fell off the slide and broke %s head!",
    "Little %s was playing with matches and lit %s hair on fire!",
    "Little %s drank a bottle of acid with %s lunch!",
    "Little %s caught %s hand in the paper shredder!",
    "Little %s wrestled with a stray dog and it bit %s finger off!"
};

char *boys[] = {
    "George", "John", "Nick", "Jim", "Constantine",
    "Chris", "Peter", "Paul", "Steve", "Billy", "Mike",
    "Vangelis", "Antony"
};

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

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

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

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

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

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

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

    int flag = 0;

```

```

do{
    sem_wait(&thr->kg->door);

    /* LOCK */

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

    if (thr->kg->space) { /* If there is space for entering a child*/

        ++(thr->kg->vc); /* Increasing the number of children by 1 */
        thr->kg->space--; /*So the free space is decreased by 1*/
        flag = 1; /* flag enabled*/
        if (thr->kg->space) /* If there is free space, send signal to other
children */
            sem_post(&thr->kg->door);
    }
    pthread_mutex_unlock(&thr->kg->mutex);
    /* UNLOCK */
}while(!flag); /*If there is space the door is opened*/
}

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);

    /* LOCK */
    pthread_mutex_lock(&thr->kg->mutex);
    --(thr->kg->vc); /*decrease the number of children by
1*/

    thr->kg->space++; /*So increase the space*/
    if(thr->kg->space > thr->kg->ratio)
        pthread_cond_broadcast(&thr->kg->condVar); /*if there is enough space
teacher is allowed to leave */
    else
        sem_post(&thr->kg->door); /*else open the door for children */
    pthread_mutex_unlock(&thr->kg->mutex);
    /* UNLOCK */
}

```

```

}

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);

    /* LOCK */

    pthread_mutex_lock(&thr->kg->mutex);
    ++(thr->kg->vt);          /*Increasing teacher by 1*/
    thr->kg->space += thr->kg->ratio;      /* So increasing the space by a ratio */
    sem_post(&thr->kg->door);           /* and open a door */
    pthread_mutex_unlock(&thr->kg->mutex);

    /* UNLOCK */
}

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);

    /* LOCK */

    pthread_mutex_lock(&thr->kg->mutex);
    while(thr->kg->space < thr->kg->ratio) pthread_cond_wait(&thr->kg->condVar, &thr-
>kg->mutex); //wait until teacher can leave
    --(thr->kg->vt);          /* decrease the number of teacher */
    thr->kg->space -= thr->kg->ratio;      /* and in turn the ratio */
    pthread_mutex_unlock(&thr->kg->mutex);

    /* UNLOCK */
}

/*

```



```

    * 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;
    kg->space = 0;

    ret = pthread_mutex_init(&kg->mutex, NULL);
    if (ret) {
        perror_thread(ret, "pthread_mutex_init");
        exit(1);
    }
    sem_init(&kg->door, 0, 0);
    pthread_cond_init(&kg->condVar, NULL);
    /* ... */
    /*
     * 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_thread(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;  
}
```

Έξοδος εκτέλεσης προγράμματος:

Η έξοδος του προγράμματος είναι μια τυχαία αλυσίδα περιπτώσεων μεταξύ αυτών είναι:

- Είσοδος παιδιού
- Έξοδος παιδιού
- Είσοδος καθηγητή
- Έξοδος καθηγητή

Ένα τυπικό δείγμα είναι το παρακάτω:

Thread 6 [Child]: Entering.

THREAD 6: CHILD ENTER

Thread 6 [Child]: Entered.

Thread 6: Teachers: 20, Children: 21

Thread 20: Teachers: 20, Children: 21

Thread 20 [Child]: Entering.

THREAD 20: CHILD ENTER

Thread 20 [Child]: Entered.

Thread 20: Teachers: 20, Children: 22

Thread 23: Teachers: 20, Children: 22

Thread 23 [Teacher]: Entering.

THREAD 23: TEACHER ENTER

Thread 23 [Teacher]: Entered.

Thread 23: Teachers: 21, Children: 22

Thread 38: Teachers: 21, Children: 22

Thread 38 [Teacher]: Entering.

THREAD 38: TEACHER ENTER

Thread 38 [Teacher]: Entered.

Thread 38: Teachers: 22, Children: 22

Thread 34 [Teacher]: Exiting.

THREAD 34: TEACHER EXIT

Thread 34 [Teacher]: Exited.

Thread 38 [Teacher]: Exiting.

THREAD 38: TEACHER EXIT

Thread 38 [Teacher]: Exited.

Thread 33 [Teacher]: Exiting.

THREAD 33: TEACHER EXIT

Thread 33 [Teacher]: Exited.

Thread 28 [Teacher]: Exiting.

THREAD 28: TEACHER EXIT

Thread 28 [Teacher]: Exited.

Thread 34: Teachers: 18, Children: 22

Thread 34 [Teacher]: Entering.

THREAD 34: TEACHER ENTER

Thread 34 [Teacher]: Entered.

Thread 34: Teachers: 19, Children: 22

Thread 35: Teachers: 19, Children: 22

Thread 35 [Teacher]: Entering.

THREAD 35: TEACHER ENTER

Thread 35 [Teacher]: Entered.

Thread 35: Teachers: 20, Children: 22

Thread 32 [Teacher]: Exiting.

THREAD 32: TEACHER EXIT

Thread 32 [Teacher]: Exited.

Thread 36 [Teacher]: Exiting.

THREAD 36: TEACHER EXIT

Thread 36 [Teacher]: Exited.

Ερωτήσεις:

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

Στο σχήμα συγχρονισμού που υλοποιήθηκε η είσοδος και έξοδος εξυπηρετείται τυχαία ανεξάρτητα του χρόνου αναμονής, αλλά σύμφωνα με την σειρά εκτέλεσης των νημάτων.

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.

Το παραπάνω condition εμφανίζεται και στον έλεγχο ορθότητας, επομένως το πρόγραμμα δεν παραβιάζει ποτέ την τυχαιότητα καθώς δεν μπλοκάρει καμία είσοδο/έξοδο του νηπιαγωγείου.