



National Technical University of Athens

School of Electrical and Computer Engineering

Λειτουργικά Συστήματα Υπολογιστών (Τμήμα 1)

Άσκηση 3: Συγχρονισμός

ΧΡΙΣΤΟΦΟΡΟΣ ΒΑΡΔΑΚΗΣ (03118883)

ΓΕΩΡΓΙΟΣ ΛΥΜΠΕΡΑΚΗΣ (03118881)

oslaba69

23 Μαΐου 2021

Άσκηση 1.1 : Συγχρονισμός σε υπάρχοντα κώδικα

Στην πρώτη άσκηση καλούμαστε να συγχρονίσουμε δύο νήματα τα οποία αυξάνουν και μειώνουν κατά μία μονάδα για N φορές το περιεχόμενο μιας μεταβλητής που έχει αρχικά τη μηδενική τιμή και την μοιράζονται και τα δύο νήματα.

Αρχικά, πριν προβούμε στον κώδικα του προγράμματος χρησιμοποιούμε το δοθέν Makefile ώστε να μεταγλωττίσουμε το πρόγραμμα και παρατηρούμε την δημιουργία δύο Object Files και δύο εκτελέσιμων (executable) αρχείων , ***simplesync-atomic.o*** , ***simplesync-mutex.o*** τα και αντίστοιχα τα ***simplesync-atomic*** , ***simplesync-mutex*** .

Για να μπορέσουμε να ερμηνεύσουμε το παραπάνω γεγονός επισκεπτόμαστε το Makefile όπου παρατηρούμε ότι κατά την πληκτρολόγηση της εντολής που θα μεταγλωττίσει το αρχείο κώδικα C υπεισέρχεται και μία από τις δύο εκφράσεις ***-DSYNC_ATOMIC*** , ***-DSYNC_MUTEX*** .

Τώρα επισκεπτόμαστε το αρχείο ***simplesync.c*** το οποίο περιλαμβάνει στην αρχή του τις παρακάτω γραμμές κώδικα.

```
#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
```

Παρατηρούμε ότι το πρόγραμμα κατά τη διαδικασία της μεταγλώττισης ελέγχει ποια από τις δύο παραμέτρους του έχει δώσει ο χρήστης ώστε να δημιουργήσει το κατάλληλο Object File . Ελέγχει με την χρήση του XOR Bitwise Operator την περίπτωση που κάποιος χρήστης έχει πληκτρολογήσει και τις δύο παραμέτρους και εκτυπώνει ανάλογο μήνυμα σφάλματος και κατάλληλη οδηγία για το πλήθος και το είδος των απαιτούμενων ορισμάτων.

Για να μπορέσουμε να λύσουμε το πρόβλημα του συγχρονισμού των δύο διεργασιών, δηλαδή να επιτύχουμε αμοιβαίο αποκλεισμό στο κρίσιμο τμήμα του κώδικα χρησιμοποιούμε δύο διαφορετικές ιδέες τα spinlocks και τις atomic operations .

Πιο συγκεκριμένα ,για τη μεν πρώτη λύση εφαρμόζονται οι συναρτήσεις

```
int pthread_mutex_lock(pthread_mutex_t *mutex) ;  
int pthread_mutex_unlock(pthread_mutex_t *mutex) ;
```

Ενώ για τη δεύτερη οι

```
type __sync_add_and_fetch (type *ptr, type value, ...);  
type __sync_sub_and_fetch (type *ptr, type value, ...);
```

Οπότε ,το περιεχόμενο του Makefile και ο κώδικας του αρχείου είναι παρακάτω μαζί και με τις εξόδους στο τερματικό μας από την εκτέλεση των δύο executable αρχείων.

Περιεχόμενα 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

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

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

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

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

simplesync-mutex.o: simplesync.c
    $(CC) $(CFLAGS) -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c

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

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

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

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

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

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

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

Αρχείο simplesync .c

```
/*
 * 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 mut;

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);

            //    ++(*ip);

        } else {
            pthread_mutex_lock(&mut);
            ++(*ip);
            pthread_mutex_unlock(&mut);
        }
    }
    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_sub_and_fetch (ip, 1);

            //    --(*ip);

        } else {
            pthread_mutex_lock(&mut);

            --(*ip);
            pthread_mutex_unlock(&mut);
        }
    }
    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;

    pthread_mutex_init(&mut, NULL);

    /*
     * Create threads
     */
    ret = pthread_create(&t1, NULL, increase_fn, &val);
    if (ret) {
        perror_thread(ret, "pthread_create");
        exit(1);
    }
    ret = pthread_create(&t2, NULL, decrease_fn, &val);
    if (ret) {
        perror_thread(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 Εκτελέσιμου *simplesync-mutex*

```
oslaba69@os-node1:~/ALL/G/sync$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

Έξοδος Εκτελέσιμου *simplesync-atomic*

```
oslaba69@os-node1:~/ALL/G/sync$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Απάντηση Ερωτήσεων:

1.

Όταν οι υπολογισμοί γίνονται χωρίς συγχρονισμό, οι πράξεις δεν μπορούν να εκτελεσθούν παράλληλα. Αντίθετα, όταν έχουμε συγχρονισμό είτε μέσω POSIX mutexes είτε ατομικών λειτουργιών οι πράξεις περιμένουν η μία να τελειώσει η προηγούμενη. Αυτό έχει ως αποτέλεσμα την επιβράδυνση του προγράμματος. Αυτό επιβεβαιώνετε και από τις παρακάτω μετρήσεις που έγιναν με τη χρήση της εντολής time.

	ΧΩΡΙΣ ΣΥΓΧΡΟΝΙΣΜΟ	ΜΕ ΣΥΓΧΡΟΝΙΣΜΟ
atomic	real 0m0.055s user 0m0.072s sys 0m0.000s	real 0m0.269s user 0m1.004s sys 0m0.012s
mutex	real 0m0.039s user 0m0.072s sys 0m0.000s	real 0m27.360s user 0m27.264s sys 0m27.448s

2.

Οι δύο λύσεις παραπάνω επιτυγχάνουν το ίδιο αποτέλεσμα ,δηλαδή την παρουσία ενός και μοναδικού νήματος στο κρίσιμο σημείο του κώδικα όπου εκεί αλλάζει η τιμή της μεταβλητής *ip*. Ωστόσο, οι δύο υλοποιήσεις δεν χρειάζονται τον ίδιο χρόνο για την ολοκλήρωση του προγράμματος. Ειδικότερα, όπως παρατηρήθηκε στο προηγούμενο ερώτημα που αναφέρθηκαν οι χρόνοι εκτέλεσης του κάθε εκτελέσιμο παρατηρούμε ότι ο χρόνος εκτέλεσης του προγράμματος με τη χρήση των ατομικών εντολών είναι σημαντικά μικρότερος από τον αντίστοιχο χρόνο όταν χρησιμοποιούνται τα κλειδώματα.

Το παραπάνω αποτέλεσμα είναι αναμενόμενο καθώς οι ατομικές εντολές (Atomic Operations) εκμεταλλεύονται την συνεισφορά της ΚΜΕ ,δηλαδή χρησιμοποιούνται εντολές (instructions) στο επίπεδο assembly (σύγκρισης και ανταλλαγής περιεχόμενου εντολές) πράγμα που κάνει αρκετά γρήγορα τον αποκλεισμό της πληροφορίας. Από την άλλη πλευρά, τα κλειδώματα (Locks) για να «τρέξουν» στον επεξεργαστή απαιτούν αυτόν να βρίσκεται σε Kernel Mode οπότε αυτή η συνεχόμενη αλλαγή κατάστασης του επεξεργαστή καταναλώνει ενέργεια και χρόνο για την ολοκλήρωση του προγράμματος . Κλείνοντας, κατά τη διαδικασία που ένα νήμα (Thread) βρίσκεται στην κρίσιμη περιοχή του κώδικα (Critical Section) τότε οι άλλοι επεξεργαστές (αν υπάρχουν φυσικά) αν επιθυμήσουν την προσπέλαση της ίδια περιοχής μνήμης τότε

αναγκάζονται να μεταβούν σε κατάσταση αναμονής έως ότου το νήμα αποχωρήσει από την κρίσιμη περιοχή οπότε έχουμε και καθυστέρηση των υπολοίπων ΚΜΕ.

3.

Από την εντολή παρακάτω,

```
gcc -Wall -O2 -pthread -DSYNC_ATOMIC -S -g simplesync.c
```

Παίρνουμε το αρχείο `simplesync.s` το οποίο περιέχει τον ενδιάμεσο κώδικα για την υλοποίηση με χρήση ατομικών λειτουργιών.

Παρατηρούμε ότι,

Στη γραμμή που καλούμε την εντολή `__sync_add_and_fetch (ip, 1)` αντιστοιχεί η εντολή `lock addl $1, (%rbx)`.

Ενώ στην εντολή `__sync_sub_and_fetch (ip, 1)` αντιστοιχεί η εντολή `lock subl $1, (%rbx)`.

4.

Αντίστοιχη διαδικασία ακολουθούμε και για την υλοποίηση μέσω POSIX mutexes. Χρησιμοποιώντας την εντολή :

```
gcc -Wall -O2 -pthread -DSYNC_MUTEX -S -g simplesync.c
```

Παίρνουμε το αρχείο `simplesync.s` το οποίο περιέχει τον ενδιάμεσο κώδικα για την υλοποίηση με χρήση ατομικών λειτουργιών.

Παρατηρούμε ότι στη γραμμή που καλούμε την εντολή `pthread_mutex_lock(&mut)` αντιστοιχεί η εντολή

```
movl $mut, %edi  
call pthread_mutex_lock
```

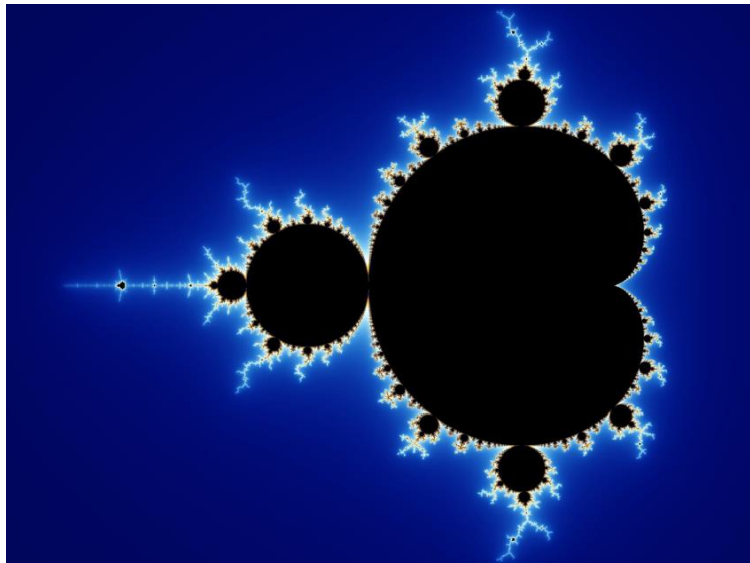
Ενώ στην εντολή

```
pthread_mutex_unlock(&mut)
```

Αντιστοιχεί η εντολή

```
movl %eax, 0(%rbp)  
call pthread_mutex_unlock
```

Άσκηση 1.2 : Παράλληλος υπολογισμός του συνόλου Mandelbrot



Στην δεύτερη άσκηση καλούμαστε να συγχρονίσουμε έναν αριθμό νημάτων ώστε να επιτύχουμε την εξαγωγή της παραπάνω εικόνας στο τερματικό μας.

Προσοχή το Makefile που παρουσιάστηκε στην προηγούμενη άσκηση καλύπτει τις απαιτήσεις όλων των ασκήσεων.

Για να μπορέσουμε να επιτύχουμε το συγχρονισμό των νημάτων – Threads θα χρησιμοποιήσουμε τους σηματοφόρους (Binary Semaphores) και τις παρακάτω συναρτήσεις που μας παρέχονται από τη βιβλιοθήκη <semaphore.h>.

Πιο συγκεκριμένα τις,

1. `int sem_post(sem_t *sem);`
2. `int sem_wait(sem_t *sem);`
3. `int sem_init(sem_t *sem, int pshared, unsigned int value);`
4. `int sem_destroy(sem_t *sem);`

Επομένως , ο κώδικας παρουσιάζεται παρακάτω

Κώδικας Αρχείου mandel.c

```
/*
 * mandel.c
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */
#include <signal.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>
#include "mandel-lib.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
 */

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

int y_chars = 50;
int x_chars = 90;

//pointer to the array of semaphores
sem_t *sema;

//struct for its thread
struct pthread_card{
    pthread_t tid;
    int thrcnt;
    int fd;
    int turn;
};

typedef struct pthread_card * thr_ptr;

//Signal Handel for the signal SIGINT
void signal_handler(int sig ){
    reset_xterm_color(1); //reset the color
    exit(1); //terminate the program
}

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

```

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

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

    int n;
    int val;

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

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

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

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

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

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

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

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

```

```

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

    thr_ptr ptr = (thr_ptr) p;    //type casting

    //take some informations from the thread struct
    int fd = ptr->fd;
    int step = ptr->thrcnt;
    int turn = ptr->turn;
    int line;

    //Let's print the image !!!
    for (line = turn; line < y_chars; line += step){
        compute_mandel_line(line , color_val);

        sem_wait(&sema[(line)%step]); //wait until one thread wakes you

        //Start of Critical Seciton
        output_mandel_line(fd , color_val); //print the your line
        //End of Critical Section

        sem_post(&sema[(line +1)% step]); //wake the thread for the next line
    }

    return NULL; //if you have printed the image then every thread
                //return from the function back to the program
}

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

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

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

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

    return p;
}

void usage(char *argv0)
{
    fprintf(stderr, "Usage: %s thread_count array_size\n\n"
        "Exactly one argument required:\n"
        "    thread_count: The number of threads to create.\n",
        argv0);
    exit(1);
}

```

```

//main function
int main(int argc, char * argv[])
{
    thr_ptr thr;          //thr ---> [ struct pthread_card ]
    int thrcnt,ret,i;

    signal(SIGINT,signal_handler);

    if (argc != 2) { //check for wrong input
        usage(argv[0]);
    }

    if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0) { //convert input string to integer
//by using "atoi"
        fprintf(stderr, "'%s' is not valid for `thread_count'\n", argv[1]);
        exit(1);
    }

    if (thrcnt > y_chars ) { //if the threads are more than the lines then compress them
        thrcnt = y_chars;
    }

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

//    printf("We are going to compute Mandelbrot using %d\n",thrcnt);

    thr = safe_malloc(thrcnt * sizeof(*thr)); //allocate space for the array of threads
//    printf("Memory Allocation for threads is OK ! \n");

    sema = safe_malloc(thrcnt * sizeof(*sema)); //allocate space for the array of semaphores
//    printf("Memory Allocation for semaphores is OK! \n");

    for (i = 1; i < thrcnt; ++i) { //for loop that initializes all apart from the first
//semaphore
        if((sem_init(&sema[i],0,0) == -1)){
            fprintf(stderr,"Error with semaphores initialization");
        }
    }

    sem_init(&sema[0],0,1); //semaphore of first thread is equal to one so it can be in
//critical section

    for (i = 0; i < thrcnt; i++) { //for-loop which create all the threads and fill their
//structures with informations

        thr[i].turn = i;
        thr[i].fd = 1;
        thr[i].thrcnt = thrcnt;

        ret = pthread_create(&thr[i].tid,NULL,compute_and_output_mandel_line,(void *)
(thr+i)); //create the thread and send it in a function (3 argument)

        if (ret<0) {
            perror_pthread(ret, "pthread_create");
            exit(1);
        }
    }
}

```

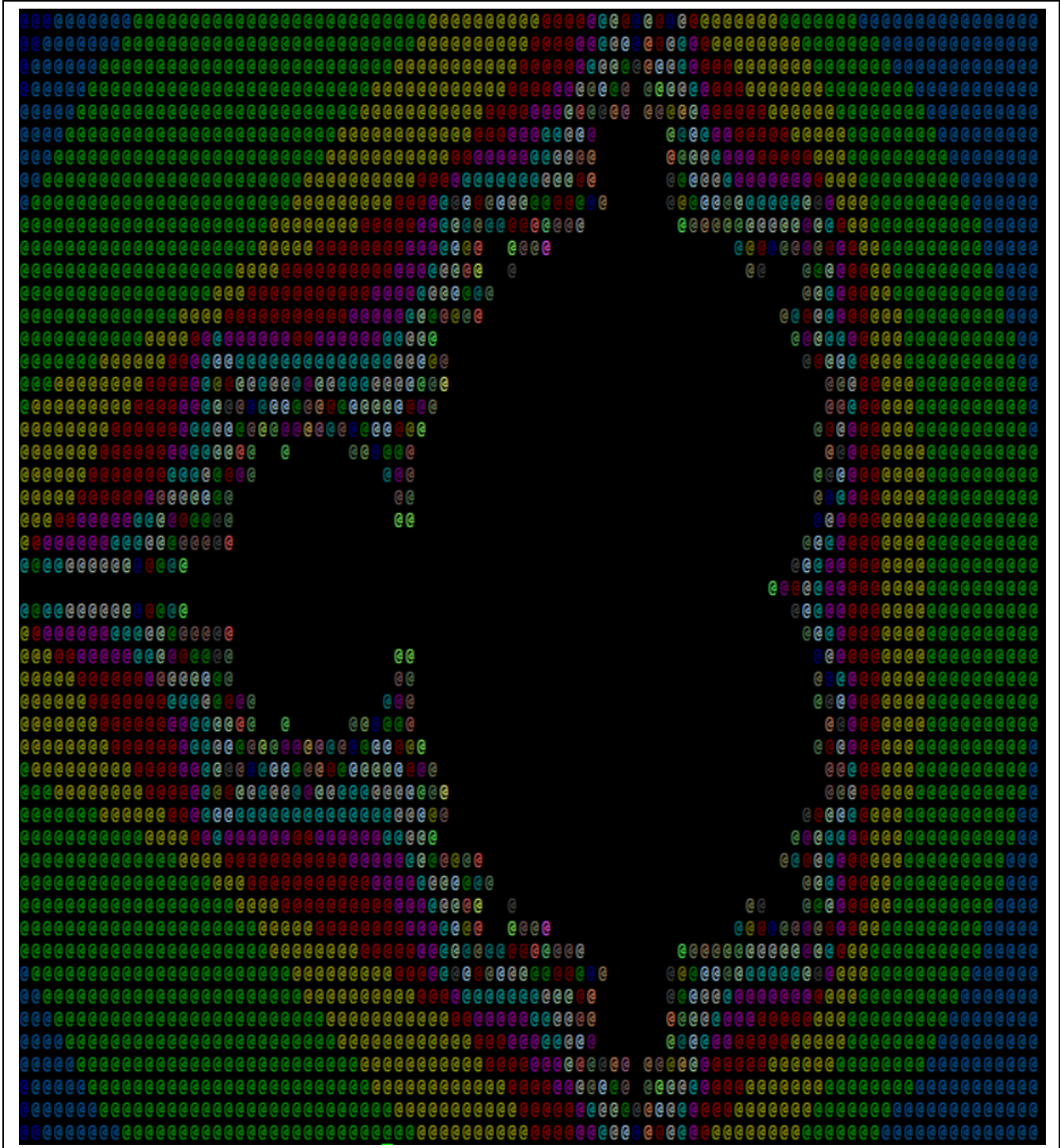
```
for (i = 0; i < thrcnt; i++) { //loop that wait every thread to finish its work
    ret = pthread_join((thr+i)->tid, NULL);

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

for (i = 0; i < thrcnt; ++i){ //destroy every semaphore
    sem_destroy(&sema[i]);
}

reset_xterm_color(1); //reset the colors back

return 0; //leave
}
```



Απάντηση Ερωτήσεων:

1.

Το παραπάνω σχήμα συγχρονισμού απαιτεί την χρήση **NTHREADS** σημαφόρων (όσα και τα νήματα που συμμετέχουν στην δημιουργία της εικόνας). Πιο συγκεκριμένα , σε κάθε νήμα αντιστοιχίζεται και ένας σημαφόρος ώστε να μπορούμε να διαχειριστούμε αποτελεσματικά τον αποκλεισμό του κρίσιμου τμήματος κώδικα ,δηλαδή την παραμονή ενός ή κανένα νήματος εκεί. Επομένως, η κυκλική εναλλαγή των νημάτων που εκτυπώνουν τη γραμμή υπολογίζουν γίνεται μέσω του προσωπικού τους σημαφόρων , όπου το νήμα που εκτυπώνει στέλνει σήμα (Signal) στο επόμενο νήμα που πρέπει να μεταβεί από την κατάσταση αναμονής (Wait) στο κρίσιμο τμήμα απομονώνοντας φυσικά όλα τα υπόλοιπα.

2.

Χρησιμοποιώντας την `cat /proc/cpuinfo` παρατηρούμε ότι οι πυρήνες του υπολογιστή που τρέξαμε είναι παραπάνω από 2 (8 μαζί με το hyperthreading) άρα επαρκούν για την παράλληλη εκτέλεση έχουμε λοιπόν τα παρακάτω αποτελέσματα:

Σειριακή εκτέλεση:

```
real 0m1.034s
user 0m0.996s
sys 0m0.012s
```

Παράλληλη εκτέλεση με 2 threads:

```
real 0m0.619s
user 0m0.992s
sys 0m0.020s
```

Όπως είναι αναμενόμενο η παράλληλη εκτέλεση είναι σημαντικά ταχύτερη από τη σειριακή.

3.

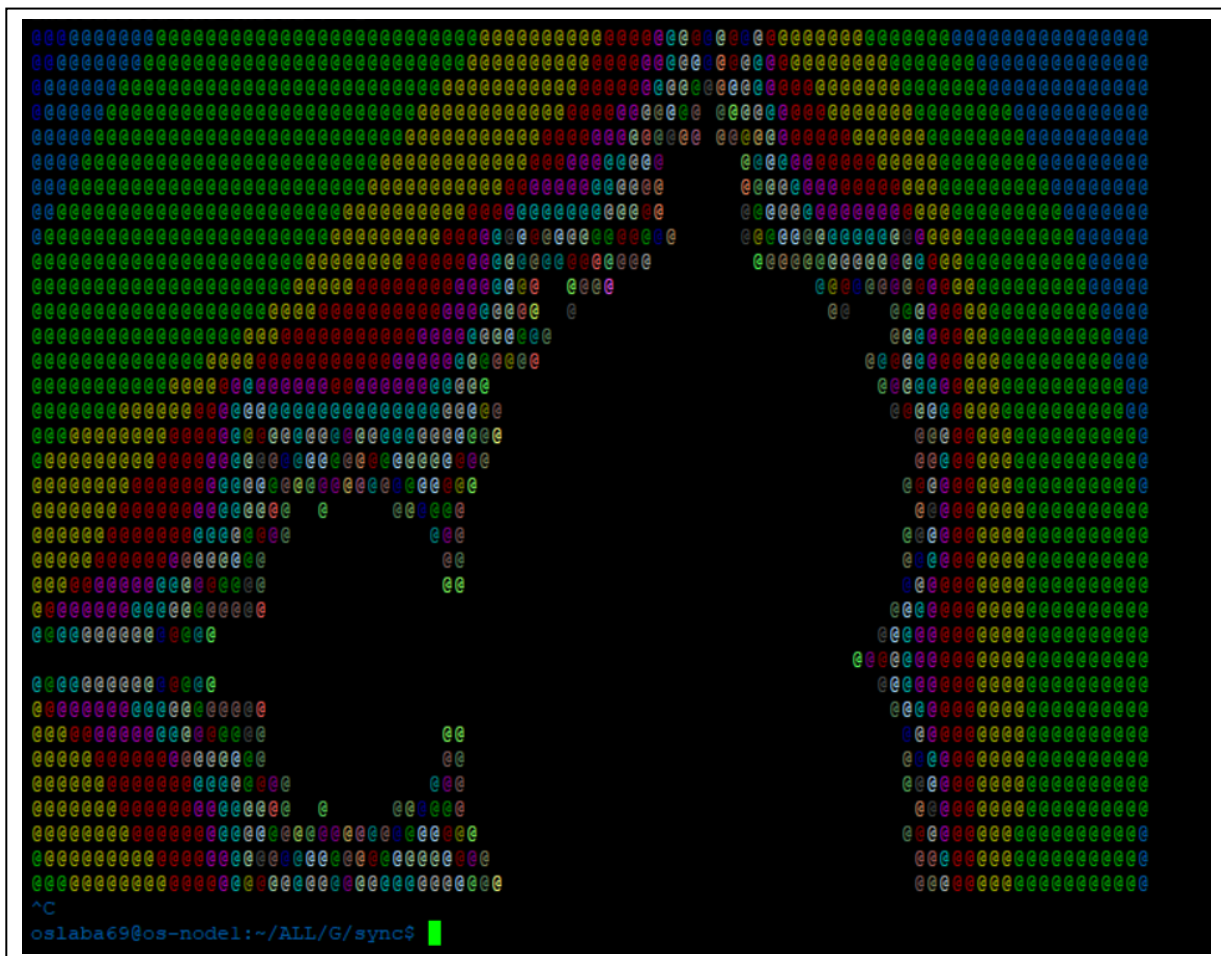
Όπως είδαμε και στο προηγούμενο ερώτημα σε πολυπύρρηνο σύστημα το πρόγραμμα εμφανίζει επιτάχυνση, τουλάχιστον όσο ο αριθμός των νημάτων είναι μικρότερος από το πλήθος των πυρήνων του συστήματος. Αξίζει να σημειωθεί ότι τα νήματα όσο το δυνατόν μικρότερο κρίσιμο τμήμα γίνεται, μιας και εκτελούν όλους τους χρονοβόρους υπολογισμούς παράλληλα και αφήνοντας μόνο την εκτύπωση , μία απλή και γρήγορη διαδικασία για το κρίσιμο τμήμα. Αυτό έχει ως αποτέλεσμα την επιτάχυνση της εκτέλεσης του προγράμματος σε σχέση με μία υλοποίηση όπου ο υπολογισμός της σειράς γίνεται μέσα στο κρίσιμο τμήμα. Μία τέτοια υλοποίηση δεν θα παρουσίαζε πρακτικά καθόλου επιτάχυνση μιας και αφού το κάθε νήμα

περιμένει την εκτέλεση του προηγούμενου για να αρχίσει τον υπολογισμό, έχουμε πρακτικά σειριακό υπολογισμό.

4.

Στην περίπτωση κατά την οποία το πρόγραμμα εκτελείται αν πατήσουμε Ctrl-C (SIGINT) τότε αποστέλλεται το σήμα τερματισμού στο πρόγραμμα και τερματίζεται η λειτουργία του . Αυτό βέβαια προξενεί το πρόβλημα της διατήρησης του χρώματος του χαρακτήρα που εκτυπωνόταν τη στιγμή που έγινε η διακοπή και στους μελλοντικούς .

Δηλαδή έχουμε το παρακάτω αποτέλεσμα,



Για να μπορέσουμε να αντιμετωπίσουμε το παραπάνω πρόβλημα θα κάνουμε χρήση της κλήσης συστήματος

```
sighandler_t signal(int signum, sighandler_t handler);
```

Όπου το σήμα που θέλουμε να ασχοληθούμε είναι το SIGINT και για όταν αποστέλλεται το συγκεκριμένο δημιουργήσαμε έναν Signal Handler ο οποίος θα επαναφέρει το χρώμα του τερματικού στην αρχική μορφή και κατόπιν θα τερματίζει το πρόγραμμα .

Ειδικότερα, οι μόνες αλλαγές που έγιναν στον παραπάνω κώδικα επισημαίνονται στο παρακάτω πλαίσιο.

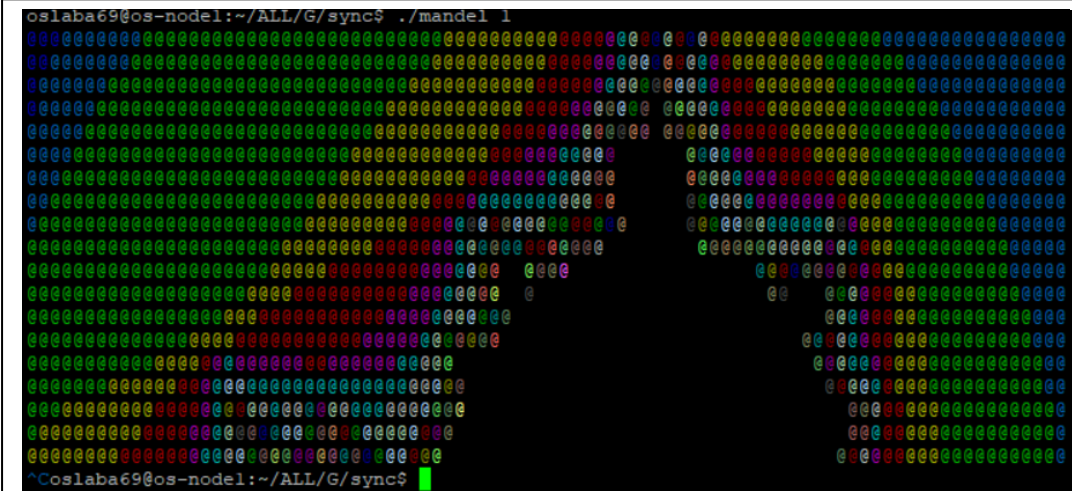
```
//Signal Handel for the signal SIGINT (Ctrl + C)
void signal_handler(int sig ){
    reset_xterm_color(1); //reset the color
    exit(1); //terminate the program
}

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

    signal(SIGINT,signal_handler);

    //Other Code
}
```

Επομένως, μετά τις αλλαγές ξανατρέχοντας το πρόγραμμα παίρνουμε το παρακάτω (ζητούμενο) αποτέλεσμα.



The screenshot shows a terminal window with the command `./mandel 1` executed. The output is a dense, colorful fractal pattern, likely a Mandelbrot set, rendered using various colors (red, green, blue, yellow, magenta, cyan) on a black background. The pattern is symmetrical and fills most of the terminal area. The prompt `^Coslaba69@os-nodel:~/ALL/G/sync$` is visible at the bottom.

Άσκηση 2.1 : Επίλυση προβλήματος συγχρονισμού

Στην άσκηση αυτή καλούμαστε να υλοποιήσουμε το σχήμα συγχρονισμού ενός νηπιαγωγείου. Πιο συγκεκριμένα πρέπει η αναλογία παιδιών – δασκάλων να μην υπερβαίνει ένα όριο, διότι εάν τα παιδιά που βρίσκονται στο νηπιαγωγείο είναι περισσότερα από όσα αντιστοιχούν σε κάθε δάσκαλο, υπάρχει περίπτωση να πάθει σοβαρή ζημιά κάποιο παιδί.

Έτσι πρέπει να μην επιτρέπεται η είσοδος στο νηπιαγωγείο σε παραπάνω παιδιά από ότι ο αριθμός των δασκάλων επί την δοσμένη αναλογία, αλλά ούτε και να επιτρέπεται η έξοδος από το νηπιαγωγείο αν μετά την έξοδο του δασκάλου η αναλογία παιδιών – δασκάλων δεν είναι σωστή. Έτσι πρέπει διαρκώς να ισχύει :

$$\# \text{Παιδιών} \leq \# \text{Δασκάλων} * \text{Αναλογία}$$

Προγραμματιστικά αυτό υλοποιείται με τη χρήση νημάτων όπου κάθε νήμα αντιστοιχεί είτε σε ένα παιδί είτε σε ένα δάσκαλο, ενός struct που διαχειρίζεται την ολική κατάσταση του νηπιαγωγείου καθώς και μία σειρά από βοηθητικές συναρτήσεις (π.χ. `verify()` , `bad_things()`). Ο συγχρονισμός επιτυγχάνετε με την χρήση mutexes, καθώς και μίας conditional variable που επιτρέπει η απαγορεύει την είσοδο και την έξοδο των παιδιών και των δασκάλων αντίστοιχα, ανάλογα με την κατάσταση του νηπιαγωγείου.

Ο συνολικός κώδικας φαίνεται παρακάτω:

Κώδικας Αρχείου kgarten.c

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

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

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

/* A virtual kindergarten */
struct kgarten_struct {

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

    pthread_cond_t cond;//condition variable
    /*
     * 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);
    }

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

    pthread_mutex_lock(&thr->kg->mutex);
    //critical section
    int r = thr->kg->ratio;
    while (thr->kg->vt*r < (thr->kg->vc+1)) { //Teachers * ratio < Children
        pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
    }

    ++(thr->kg->vc);
    //end of critical section
    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);
    //critical section
    --(thr->kg->vc);

    pthread_cond_broadcast(& thr->kg->cond);
    //end of critical section
    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);
    //critical section
    ++(thr->kg->vt);

    pthread_cond_broadcast(& thr->kg->cond);
    //end of critical section
    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);
    //critical section
    int r = thr->kg->ratio;

    while (thr->kg->vc > (thr->kg->vt - 1) * r ){
        pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
    }
    --(thr->kg->vt);
    //end if critical section
    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);
    }
    /*
    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;
}

```

```
oslab69@os-node1:~/ALL/C/Ex3/sync$ ./kgarten 10 7 3
Thread 0 of 10. START.
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
Thread 2 of 10. START.
Thread 2 [Child]: Entering.
THREAD 2: CHILD ENTER
Thread 1 of 10. START.
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 4 of 10. START.
Thread 4 [Child]: Entering.
THREAD 4: CHILD ENTER
Thread 5 of 10. START.
Thread 3 of 10. START.
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER
Thread 5 [Child]: Entering.
Thread 6 of 10. START.
Thread 7 of 10. START.
Thread 7 [Teacher]: Entering.
THREAD 7: TEACHER ENTER
Thread 6 [Child]: Entering.
Thread 0 [Child]: Entered.
Thread 2 [Child]: Entered.
Thread 8 of 10. START.
THREAD 6: CHILD ENTER
Thread 7 [Teacher]: Entered.
Thread 9 of 10. START.
Thread 9 [Teacher]: Entering.
THREAD 9: TEACHER ENTER
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
THREAD 5: CHILD ENTER
    Thread 0: Teachers: 1, Children: 2
Thread 4 [Child]: Entered.
    Thread 4: Teachers: 1, Children: 3
    Thread 2: Teachers: 1, Children: 3
    Thread 7: Teachers: 1, Children: 3
Thread 9 [Teacher]: Entered.
```

1.

Στην υλοποίησή μας χρησιμοποιήσαμε μία μεταβλητή κατάστασης. Για τον λόγο αυτό, όταν κάποιος δάσκαλος επιχειρεί να φύγει από το νηπιαγωγείο υπάρχει στο διάστημα αυτό περίπτωση, εάν φυσικά υπάρχουν επαρκείς θέσεις, να εισέλθουν παιδιά στο νηπιαγωγείο. Δηλαδή εάν σε μία στιγμή δίνετε να βγει από το νηπιαγωγείο δάσκαλος και να μπει στο νηπιαγωγείο παιδί είναι τυχαίο το πιο από τα δύο θα συμβεί. Παρόλα αυτά, όποιο νήμα ξυπνήσει και μπει στο κρίσιμο σημείο θα κλειδώσει το `mutex`, οπότε δεν υπάρχει περίπτωση να εκτελεσθούν και τα δύο αιτήματα. Έτσι επιβεβαιώνετε ότι δεν γίνεται να βρεθεί το νηπιαγωγείο σε επικίνδυνη κατάσταση.

2.

Στον κώδικα μας υπάρχουν `race conditions` τα οποία όμως επιλύονται με αποτέλεσμα να μην αποτελούν κίνδυνο σφάλματος.

Ποιο συγκεκριμένα,

Κατά την κλήση της `verify` καθώς και κατά τον έλεγχο στην είσοδο και στην έξοδο των νημάτων, υπάρχει περίπτωση να γίνει αλλαγή στις μεταβλητές την ώρα που υπολογίζετε η ορθότητα της κατάστασης του νηπιαγωγείου. Όμως η χρήση του `mutex` δεν επιτρέπει σε κανένα άλλο νήμα να μεταβάλει τις τιμές όσο η συνάρτηση υπολογίζει το αποτέλεσμα.

Άλλη μία `race condition` υπάρχει στη περίπτωση που ένα παιδί εισέρχεται και ένας δάσκαλος εξέρχεται από το νηπιαγωγείο. Παρόλα αυτά το γεγονός ότι έχουμε κοινή `conditional variable` για τους δασκάλους και τα παιδιά διασφαλίζει ότι εάν βρίσκονται σε κατάσταση `pthread_cond_wait` ένας δάσκαλος και ένα παιδί δεν υπάρχει περίπτωση να ξυπνήσουν και τα δύο νήματα. Έτσι αποφεύγεται το `race condition`. Επίσης χάρη στο `while loop` που περιβάλλει την `pthread_cond_wait` είναι βέβαιο πως ακόμα και αν ξυπνήσει από την `wait` κάποιο νήμα που βρισκόταν σε αναμονή, θα επανεξετάσει τις συνθήκες πριν συνεχίσει την εκτέλεση του κρίσιμου σημείου του.

Προαιρετικές Ερωτήσεις :

1.

Παρακάτω παρουσιάζεται σε C η υλοποίηση των σηματοφόρων με τη χρήση παρακολουθητών.

```
struct Semaphore {  
  
    int counter;  
    Queue<Process_PID> q;  
  
    int P(Semaphore s){  
        s.value--;  
  
        if(s.value < 0){  
            q.push(process_pid);  
            Block();  
        }  
  
        else  
            return;  
    }  
  
    int V(Semaphore s){  
        s.value++;  
  
        if(s.value <= 0){  
            process_pid = q.pop();  
            Wake_Up(process_pid);  
        }  
        else  
            return;  
    }  
  
};
```

2.

Παρατηρούμε ότι παρόλο που κάθε νήμα τρέχει την `rand()` η οποία είναι φτιαγμένη για την παραγωγή τυχαίων αριθμών, το αποτέλεσμα της σε κάθε νήμα είναι ίδιο. Αυτό συμβαίνει διότι η `rand()` παράγει πραγματικά τυχαίους αριθμούς, πράγμα αδύνατο σε ένα ντετερμινιστικό σύστημα όπως ο υπολογιστής, αλλά ένα ψευδοτυχαίο αριθμό σε κάθε κλήση της με τη χρήση κάποιας βοηθητικής συνάρτησης. Μάλιστα αν δε συνοδεύεται από την `srand()` το αποτέλεσμα της είναι ίδιο σε κάθε εκτέλεση του προγράμματος. Αυτό που κάνει η `srand()` είναι να δίνει ένα όρισμα στην βοηθητική συνάρτηση της `rand()` με αποτέλεσμα είτε να παράγει συγκεκριμένους αριθμούς αν το όρισμα είναι σταθερό (πχ για την χρήση ίδιων τυχαίων αριθμών σε πολλά προγράμματα ή μεταξύ των μεταγλωττίσεων) ή αν το όρισμα αυτό είναι κάτι τυχαίο (όπως η χρονική στιγμή που γίνεται η κλήση της `srand`) να προσδίδει περισσότερη τυχειότητα στη συνάρτηση. Όμως στη συγκεκριμένη περίπτωση, επειδή η εκτέλεση της `srand` γίνεται στον κορμό του προγράμματος κάθε νήμα έχει το ίδιο όρισμα άρα οι τυχαίοι αριθμοί που παράγει θα είναι οι ίδιοι.