ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Λειτουργικά Συστήματα

7ο Εξάμηνο

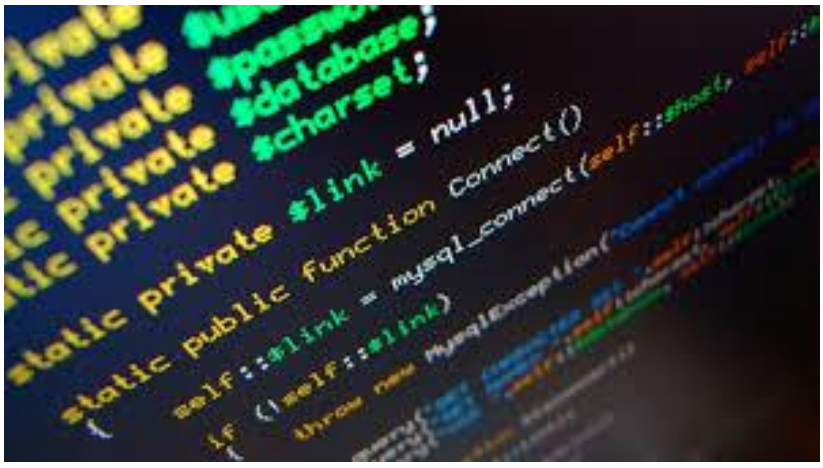
Άσκηση 3

Συγχρονισμός

Εργαστηριακή Ομάδα Δ01

Σταυρακάκης Δημήτριος
ΑΜ: 03112017

Αθανασίου Νικόλαος
ΑΜ: 03112074



Άσκηση 3:
Συγχρονισμός

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Στο συγκεκριμένο ερώτημα αυτής της εργαστηριακής άσκησης καλούμαστε να επιτύχουμε το συγχρονισμό δύο νημάτων που τρέχουν ταυτόχρονα με βάση το δοσμένο κώδικα simplesync.c. Στην ουσία, το ένα νήμα καλείται να αυξήσει τη μεταβλητή val κατά N (+1 σε κάθε επανάληψη) και το άλλο να τη μειώσει κατά N (-1 σε κάθε επανάληψη). Επομένως, επειδή η μεταβλητή val αρχικοποιείται στο 0, μετά από το συγχρονισμό των δύο αυτών διαδικασιών η τιμή της πρέπει πάλι να είναι 0. Όμως, αρχικά τα νήματα , όπως μας δίνονται, δεν είναι συγχρονισμένα με αποτέλεσμα να παρεμβαίνει το ένα στην εκτέλεση του άλλου και η μεταβλητή val να καταλήγει να έχει απροσδόκητες τιμές στο τέλος του προγράμματος. Για να επιλύσουμε αυτό το πρόβλημα ακόλουθήσαμε χρησιμοποιήσαμε τον κώδικα που μας δίνεται και τον προσαρμόσαμε κατάλληλα ώστε να επιτύχουμε το συγχρονισμό με 2 διαφορετικούς τρόπους:

1. με χρήση ατομικών λειτουργιών του GCC
2. με χρήση POSIX mutexes

Παρατηρούμε πως με την εντολη make παράγονται 2 εκτελέσιμα για τον κώδικα μας. Ένα το simplesync-atomic και ένα το simplesync-mutex (το όνομα του καθενός υποδηλώνει το πως επιτυγχάνεται ο συγχρονισμός).
Το ποιον από τους 2 τρόπους συγχρονισμού έχουμε ορίζεται στον κώδικα μας από τις εντολές:

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

Ανάλογα με την τιμή του USE_ATOMIC_OPS επιλέγεται και η κατάλληλη if που θα εκτελέσει το πρόγραμμά μας. Το ποιο εκτελέσιμο θα παραχτεί κρίνεται με βάση τις παρακάτω εντολές που βρίσκονται στο makefile:

```
 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
```

Έτσι παράγονται τα 2 διαφορετικά εκτελέσιμα (με linking με τα .o files που παράγονται απο τις 2 παραπάνω εντολές του makefile).

Ο τροποποιημένος κώδικας που υλοποιεί το συγχρονισμό είναι ο ακόλουθος:

```
#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 lock;
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(&lock);
			/* ... */
			/* You cannot modify the following line */
			++(*ip);
			/* ... */
			pthread_mutex_unlock(&lock);
		}
	}
	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(&lock);
			/* ... */
			/* You cannot modify the following line */
			--(*ip);
			/* ... */
			pthread_mutex_unlock(&lock);
		}
	}
	fprintf(stderr, "Done decreasing variable.\n");

	return NULL;
}


int main(int argc, char *argv[])
{
	int val, ret, ok;
	pthread_t t1, t2;

	/*
	 * Initial value
	 */
	val = 0;

	/*
	 * Create threads
	 */
	ret = pthread_create(&t1, NULL, increase_fn, &val);
	if (ret) {
		perror_pthread(ret, "pthread_create");
		exit(1);
	}
	ret = pthread_create(&t2, NULL, decrease_fn, &val);
	if (ret) {
		perror_pthread(ret, "pthread_create");
		exit(1);
	}
	/*
	 * Wait for threads to terminate
	 */
	ret = pthread_join(t1, NULL);
	if (ret)
		perror_pthread(ret, "pthread_join");
	ret = pthread_join(t2, NULL);
	if (ret)
		perror_pthread(ret, "pthread_join");
```

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

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

	return ok;
}
```

Στον παραπάνω κώδικα, αν επιλέξουμε να επιτύχουμε το συγχρονισμό με gcc atomic operations χρησιμοποιούμε τις εντολές __sync_add_and_fetch(ip,1) για την αύξηση και αντίστοιχα __sync_sub_and_fetch(ip,1) για τη μείωση της μεταβλητής val. Και οι 2 εντολές δρουν σε επίπεδο υλικού. Οι 2 αυτές εντολές εξασφαλίζουν πως πρώτα θα αποθηκεύσουν την τιμής της μεταβλητής στη θέση μνήμης από όπου την πήραν, αφού τελειώσει η επιθυμητή λειτουργία, και μετά θα είναι δυνατή η τροποποίησή της από άλλον στο πρόγραμμα. Έτσι, όταν πραγματοποιείται η άυξηση κατά ένα δε μπορεί να μειωθεί η μεταβλητή την ίδια στιγμή. Περιμένει δηλαδή να ολοκληρωθεί η άυξηση για να γίνει η μείωση. Αντίστοιχα, το ίδιο συμβαίνει και για τη μείωση. Έτσι επιτυγχάνεται ο συγχρονισμός. Η έξοδος του εκτελέσιμου simplesync-atomic είναι η ακόλουθη όπως αναμενόταν:

```
oslabd01@zakynthos:~/Askhsh3/sync$ ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Αν τώρα δεν επιλέξουμε ο συγχρονισμός να γίνει με gcc atomic operations, αυτός θα επιτευχθεί με χρήση POSIX mutexes. Στην περίπτωση αυτή, όταν ένα thread επιχειρεί αύξηση(ή αντίστοιχα μείωση) της μεταβλητής val, κλειδώνει αρχικά , ώστε το άλλο thread να μη μπορεί να μπει στο κρίσιμο τμήμα του κώδικα του, έως ότου το πρώτο να ολοκληρώσει την ενέργεια του, μετά από την οποία ξεκλειδώνει και επιτρέπει στο έταιρο thread να μπει και να εκτελέσει με τη δική του σειρά την ενέργειά του (αφού και αυτό κλειδώσει κοκ.). Έτσι επιτυγχάνεται ο συγχρονισμός με κλειδώματα.
Η έξοδος του εκτελέσιμου simplesync-mutex είναι η ακόλουθη όπως αναμενόταν:

```
oslabd01@zakynthos:~/Askhsh3/sync$ ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

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;

	Simplesync-atomic	**Simplesync-mutex**
Με συγχρονισμό	0.144 sec	0.766 sec
Χωρίς συγχρονισμό	0.044 sec	0.044 sec

Με βάση τους παραπάνω χρόνους, μπορούμε να συμπεράνουμε πως με το να κάνουμε συγχρονισμό η εκτέλεση του προγάμματος είναι πιο αργή. Αυτό είναι λογικό καθώς είτε με χρήση gcc atomic operations είτε με χρήση POSIX mutexes, το ένα νήμα αναμένει το άλλο να ολοκληρώσει μια ενέργεια και αυτό έχει σαν αποτέλεσμα ο χρόνος εκτέλεσης να αυξάνεται σε σχέση με την περίπτωση που δεν έχουμε συγχρονισμό.

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

Ρίχνοντας πάλι μια ματιά στους παραπάνω χρόνους όπως μας τους έδωσε η εντολή time(1), παρατηρούμε ότι η μέθοδος με χρήση ατομικών λειουργιών του GCC είναι γρηγορότερη σε σχέση με αυτή των POSIX mutexes. Ο λόγος που συμβαίνει αυτό είναι κρυμμένος στην assembly που παράγεται και αντιστοιχεί σε κάθε εκτελέσιμο. Στη περίπτωση των ατομικών λειτουργιών , η εντολή αύξησης (ή μείωσης αντίστοιχα) μεταφράζεται σε μία μονο εντολή assembly ενώ στην περίπτωση των POSIX mutexes ο κώδικας που επιτυγχάνει το συγχρονισμό μεταφράζεται σε μια σειρά εντολών κώδικα assembly κάνοντας έτσι την εκτέλεση του προγράμματος πιο αργή, αφού η πρόσβαση στο κρίσιμο τμήμα κάθε νήματος, που γίνεται μεγάλο αριθμό φορών για το κάθε νήμα, θα μας οδηγήσει στο να απαιτούνται πολύ περισσότεροι κύκλοι μηχανής στην περίπτωση χρήσης των POSIX mutexes σε σχέση με το συγχρονισμο με χρήση των GCC Atomic Operations. Έτσι αιτιολογείται η σημαντική διαφορά στο χρόνο εκτέλεσης.

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 γ α 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.

Οι ατομικές εντολές, όπως είπαμε και παραπάνω μεταφράζονται με μια εντολή assembly.

- Η εντολή __sync_add_and_fetch(ip,1), μεταφράζεται στην lock addl $1,(%ebx) .
- Η εντολή __sync_sub_and_fetch(ip,1) μεταφράζεται στην lock addl $-1,(%ebx).

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 ερωτημα.

Με την ίδια διαδικασία που ακολουθήσαμε και το προηγούμενο ερώτημα παρατηρούμε ότι στην περίπτωση χρήσης των POSIX mutexes οι εντολές assembly που παράγονται είναι οι ακόλουθες:
Για το κλείδωμα:
movl $mutex1,(%esp)
call pthread_mutex_lock
Για το ξεκλείδωμα:
movl $mutex1,(%esp)
call pthread_mutex_unlock

Έτσι γίνεται φανερό πως στην περίπτωση των mutexes θέλουμε παραπάνω εντολές άρα και παραπάνω κύκλους μηχανής σε σχέση με την περίπτωση των atomic operations, πράγμα που επαληθεύει και την απάντησή μας στο 2ο ερώτημα.

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Στο συγκεκριμένο ερώτημα αυτής της άσκησης, καλούμαστε να προσαρμόσουμε τον κώδικα που σχεδιάζει την ακολουθία του συνόλου Mandelbrot που μας δίνεται. Στον κώδικα που μας δίνεται οι υπολογισμοί γίνονται χωρίς καταμερισμό εργασίας σε νήματα. Εμείς θα χωρίσουμε τη δουλειά σε νήματα και θα τα συγρονίσουμε, ώστε το ένα νήμα να μην παρεμβαίνει όταν κάποιο άλλο πάει να εκτυπώσει τη γραμμή του καθώς έτσι θα είχαμε αλλοίωση του σχήματος. Για το συγχρονισμό αυτόν φτιάξαμε έναν πίνακα απο σημαφόρους, όσα είναι και τα νήματά μας. Τους αρχικοποιούμε όους, πλην του πρώτου (που τον αρχικοποιούμε σε 1για να ξεκινησει αμέσως ο υπολογισμός της πρώτης γραμμής) στο 0. Έτσι, κάθε thread, όταν τυπώσει τη γραμμή που είναι υπέυθυνο να τυπώσει, στέλνει σήμα και αυξάνει το σημαφόρο του επόμενου νήματος (sem_post) ώστε εκείνο με τη σειρά του να ξεκλειδώσει και να τυπώσει τη δική του γραμμή. Έτσι, πετυχαίνουμε το εξής:
Οι γραμμές του σχήματος εμφανίζονται 1-1 ενώ οι υπολογισμοί τους γίνονται ταυτόχρονα γλιτώνοντας έτσι χρόνο. Ο κώδικας που υλοποιεί όσα αναφέραμε είναι ο ακόλουθος:

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

#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

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

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

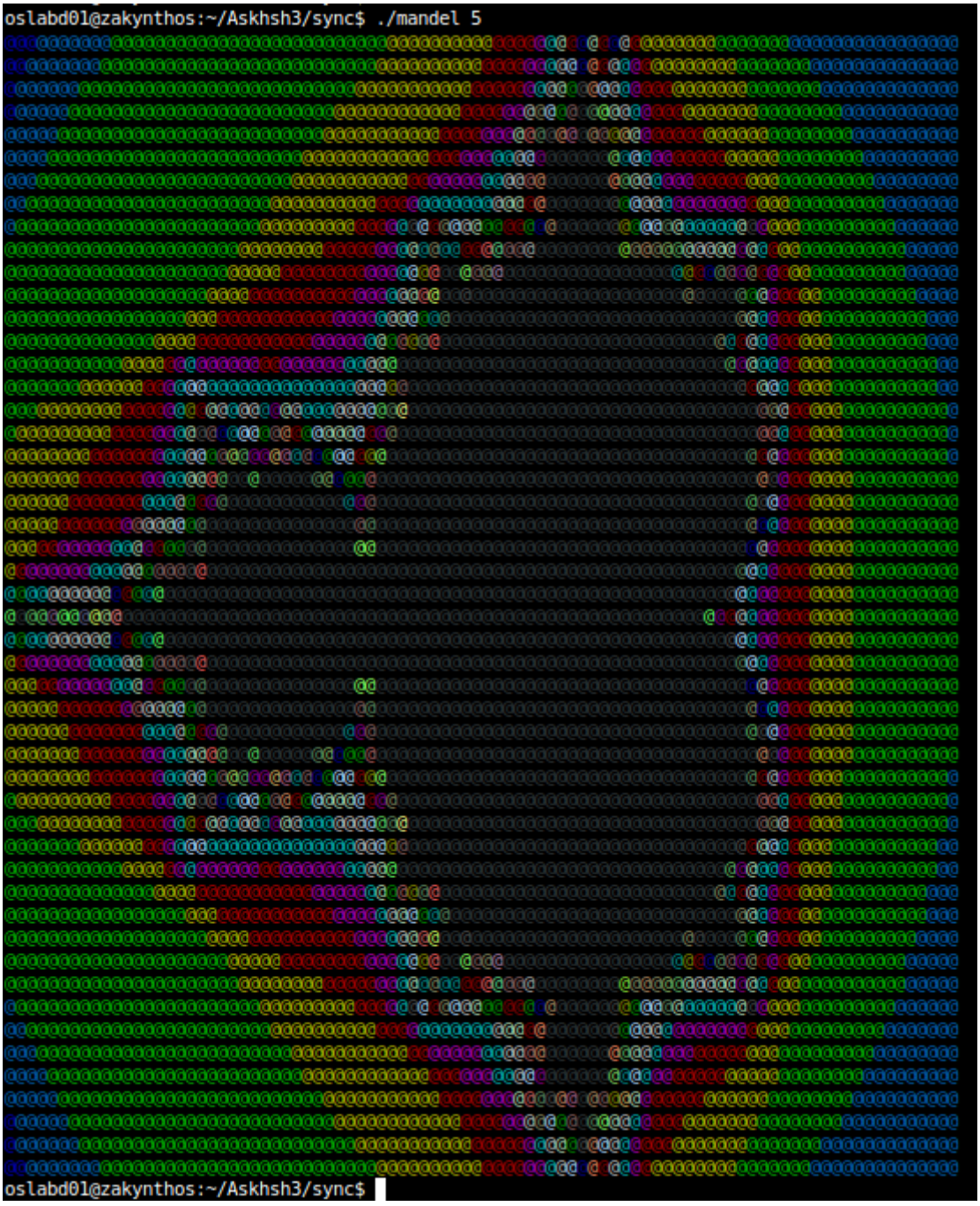
```
 xstep x ystep units wide on the complex plane. */
double xstep;
double ystep;
/*A (distinct) instance of this structure
  is passed to each thread */

struct thread_info_struct {
        pthread_t tid;  /* POSIX thread id, as returned by the library */
        int tnumber;    // px thread number 1
        int numofthrd;  // numofthreads is always the given number
};
typedef struct thread_info_struct * thrptr;

int NTHREADS;

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;
}


/*
 * 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(int fd, int line)
{
        /*
         A temporary array, used to hold color values for the line being drawn
         */
```

```
        int color_val[x_chars];

        compute_mandel_line(line, color_val);
        output_mandel_line(fd, color_val);

}
void *thread_start_fn(void *arg)
{
        int line;
        int color_val[x_chars];
        /* We know arg points to an instance of thread_info_struct */
        thrptr thread = arg;

        /*draw the Mandelbrot Set, one line at a time.
         Output is sent to file descriptor '1', i.e., standard output. */
        for (line = thread->tnumber ; line < y_chars; line += NTHREADS ) {
                compute_mandel_line(line, color_val);
                sem_wait(&sem[thread->tnumber]);
                output_mandel_line(1, color_val);
                if (thread->tnumber != NTHREADS-1)
                        sem_post(&sem[thread->tnumber + 1]);
                else
                        sem_post(&sem[0]);
        }

        return NULL;
}

int main(int argc, char *argv[])
{
        int line,  i;
        thrptr threads;
        int ret;
        xstep = (xmax - xmin) / x_chars;
        ystep = (ymax - ymin) / y_chars;

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

        if (safe_atoi(argv[1], &NTHREADS) < 0 || NTHREADS <= 0) {
                fprintf(stderr, "`%s' is not valid for `NTHREADS'\n", argv[2]);
                exit(1);
        }


        threads = safe_malloc(NTHREADS * sizeof(*threads));

        sem = safe_malloc(NTHREADS * sizeof(*sem));
```

```
	/*
	  Initialize my semaphores
	*/
	sem_init(&sem[0],0,1);
	for(line = 1 ; line < NTHREADS ; line++){
		sem_init(&sem[line], 0, 0);
	}


	for(line=0; line < NTHREADS ; line++){
		threads[line].tnumber = line;
		threads[line].numofthrd = NTHREADS;
		/* Spawn new thread */
		ret = pthread_create(&threads[line].tid, NULL, thread_start_fn, &threads[line]);
		if (ret) {
			perror_pthread(ret, "pthread_create");
			exit(1);
		}

	}

//	sem_post(&sem[0]);

	for (i = 0; i < NTHREADS; i++) {
		ret = pthread_join(threads[i].tid, NULL);
		if (ret) {
				perror_pthread(ret, "pthread_join");
			exit(1);
		}
	}

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

Αν εκτελέσουμε τώρα το παραπάνω πρόγραμμα το αποτέλεσμα που παίρνουμε είναι το ακόλουθο: (έστω ότι χρησιμοποιούμε 5 threads)



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

Για το συγχρονισμό των νημάτων στον κώδικα μας χρησιμοποιούμε έναν πίνακα σημαφόρων με μέγεθος όσα τα threads που χρησιμοποιούμε. Αρχικά θέτουμε όλους τους σημαφόρους πλην του 1ου στο 0, όπως αναφέραμε και παραπάνω. Όλα τα νήματα ξεκινούν να υπολογίζουν το καθένα τη γραμμή που του αντιστοιχεί. Το πρώτο νήμα, τυπώνει την 1η γραμμή και στέλνει σήμα sem_post στο επόμενο για να αυξήσει το σημαφόρο του , να ξυπνησει και να τυπώσει τη δική του γραμμη (ενω το 1ο τωρα υπολογίζει την επόμενη γραμμη που πιθανώς του αντιστοιχεί να τυπώσει και κανει sem_wait, περιμένοντας σήμα από άλλο για να ξεκινήσει να τυπώνει και πάλι) . Στη συνέχεια το 2ο νήμα ξυπνάει το 3ο και έτσι συνεχίζεται και τυπώνονται όλες οι γραμμές του σχήματος, ενώ οι υπολογισμοί γίνονται ταυτόχρονα γλιτώνοντας έτσι χρόνο σε σχέση με το σειριακό πρόγραμμα χωρίς νήματα. Αξίζει να πούμε ότι αν ο αριθμός των νημάτων είναι μικρότερος από τον αριθμό των

γραμμών, η αφύπνιση των νημάτων γίνεται κυκλικά, δηλαδή το Ν-οστό νήμα θα ξυπνήσει το πρώτο το οποίο τότε θα πρέπει να τυπώσει τη Ν+1 γραμμή. Αυτό συνεχίζεται έως ότου τυπωθεί όλο το σχήμα στο terminal.

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. Χρησιμοποιείστε την εντολή cat /proc/cpuinfo για να δείτε πόσους υπολογιστικούς πυρήνες διαθέτει κάποιο μηχάνημα.

Για το πρόγραμμα με παράλληλους υπολογισμούς με 2 νήματα που έχουμε υλοποιήσει τα αποτελέσματα της εντολής time σε μηχάνημα με 2 πυρήνες είναι τα ακόλουθα:

```
real    0m0.680s
user    0m1.276s
sys     0m0.020s
oslabd01@lemnos:~/Askhsh3/sync$
```

Για το σειριακό πρόγραμμα όπως αυτό μας δίνεται η εντολή time δίνει τα ακόλουθα αποτελέσματα:

```
real    0m1.316s
user    0m1.296s
sys     0m0.012s
oslabd01@lemnos:~/sync$
```

Παρατηρούμε τη διαφορά που υπάρχει με τον καταμερισμό της εργασίας σε νήματα.

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 φάσε εξόδου κάθε γραμμής που παράγεται;

Με βάση τα αποτελέσματα της εντολής time(1) παρατηρούμε ότι το υπάρχει όντως επιτάχυνση στο πρόγραμμά μας όταν αυτό εκτελείται σε διπύρηνο επεξεργαστή και χρησιμοποιεί συγχρονισμένα νήματα. Το αποτέλεσμα αυτό είναι αναμενόμενο. Αυτό γιατί, η εργασία-υπολογισμοί που γίνονται απο το σειριακό πρόγραμμα σε έναν πόρο, τώρα διαμοιράζονται σε 2 πόρους κατάλληλα και γίνονται ταυτόχρονα. Ωστόσο, για να είναι σωστό το πρόγραμμά μας πρέπει να επιτευχθεί καλά ο συγχρονισμός. Πρέπει δηλαδή το κάθε νήμα να εκτυπώνει το αποτέλεσμα των υπολογισμών του και το άλλο νήμα (ή τα άλλα νήματα αν έχουμε παραπάνω) να το περιμένει να τελειώσει για να τυπώσει μετά αυτό τη δική του γραμμή. Έτσι μόνο θα έχουμε σωστη εκτύπωση αποτελεσμάτων. Επομένως το κρίσιμο τμήμα στον κώδικα κάθε νήματος είναι η εκτύπωση της γραμμης που έχει υπολογίσει. Έτσι, μόνο σε αυτό το κομμάτι κώδικα θα έχουμε μπλοκάρισμα με χρήση των σημαφόρων. Η εκτύπωση όμως μιας γραμμής είναι αρκετά γρήγορη επομένως το μπλοκάρισμα των άλλων νημάτων θα είναι πολυ σύντομο. Αντίθετα, οι υπολογισμοί που απαιτεί η κάθε γραμμή, που είναι στην ουσία η χρονοβόρα διαδικασία που καλείται να υλοποιήσει το πρόγραμμά μας, γίνονται ταυτόχρονα από όλα τα νήματα, καθώς δεν πρέπει να περιέχονται στο κρίσιμο τμήματα, και αυτό οδηγεί στη σημαντική πτώση του χρόνου εκτέλεσης του προγράμματος. Αν βάζαμε και τους υπολογισμούς στο κρίσιμο τμήμα, το πρόγραμμά μας δε θα παρουσίαζε σημαντική διαφορά με το σειριακό αφού ο υπολογισμός κάθε γραμμής θα ακολουθούσε αφού τυπωνόταν και η προηγούμενη. (Στην ουσία οι υπολογισμοί σειριακά θα γίνονταν)

4. Τι συμβαίνει στο τερματικό αν πατήσετε Ctrl-C ενώ το πρόγραμμα εκτελείται; Σε τι κατάσταση αφήνεται, όσον αφορά το χρώμα των γραμμάτων; Πώς θα μπορούσατε να επεκτείνετε το mandel.c σας ώστε να εξασφαλίσετε ότι ακόμη και αν ο χρήστης πατήσει Ctrl-C , το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του;

Με το πάτημα του Ctrl-C, διακόπτεται αμέσως η λειτουργία του προγράμματός μας. Εμφανίζεται στο τερματικό μόνο το μέρος του συνόλου Mandelbrot που έχει προλάβει το πρόγραμμά μας να υπολογίσει και να τυπώσει μέχρι εκείνη τη στιγμή. Το τερματικό τότε περνάει σε κατάσταση αναμονής , περιμένοντας απο το χρήστη να εισάγει την επομένη εντολή. Αν πληκτρολογήσουμε έναν χαρακτήρα , θα παρατηρήσουμε ότι έχει το χρώμα του τελευταίου χαρακτήρα του συνόλου Mandelbrot που πρόλαβε να τυπωθεί. Ωστότο αυτό το “bug” μπορεί να διορθωθεί και να εξασφαλίσουμε ότι ακόμη και αν ο χρήστης πατήσει το Ctrl-C , το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του. Αυτό γίνεται αν προσθέσουμε μια ρουτίνα sighandler, καθώς το πάτημα του Ctrl-C στέλνει σήμα SIGINT για άμεσο τερματισμό. Αυτή την περίπτωση τερματισμού πρέπει να χειριστούμε. Έτσι στον handler αυτού του σήματος, απλώς θα ορίζουμε να καλείται η συνάρτηση reset_xterm_color(1) πρώτα και στη συνέχεια να τερματίζεται το πρόγραμμα. Έτσι θα επανέρχεται το τερματικό στην προηγούμενή του κατάσταση.

1.3 Επίλυση προβλήματος συγχρονισμού

Στο ερώτημα αυτό της άσκησης θα κάνουμε μια προσομοίωση ενός νηπιαγωγείου. Σε αυτό μπορούν να παρευρίσκονται παιδιά και δάσκαλοι. Ωστόσο, κάθε στιγμή πρέπει να τηρούνται οι κατάλληλες αναλογίες, οι οποίες δίνονται στο πρόγραμμά μας ως παράμετροι. Δηλαδή, κάθε δάσκαλος είναι υπεύθυνος για κάποιο συγκεκριμένο αριθμό παιδιών. Κάθε δάσκαλος και κάθε παιδί αναπαρίσταται από το πρόγραμμά μας με ένα νήμα. Κάθε δάσκαλος και παιδί εχει τη δυνατότητα να μπαίνει και να βγαίνει στο νηπιαγωγείο. Ωστόσο, κάθε στιγμή πρέπει να τηρούνται οι κατάλληλες προυποθέσεις, τις οποίες τσεκάρει η συνάρτηση verify. Τα κρίσιμα τμήματα του προγράμματος είναι προφανώς τα σημεία όπου μπαίνουν-βγαίνουν δάσκαλοι-παιδιά. Αυτά τα σημεία ελέγχουμε με μεταβλητές κατάστασης και κλειδώματα. Πιο συγκεκριμένα, όταν ένα παιδί επιχειρεί να εισέλθει στο νηπιαγωγείο πρέπει να είναι σίγουρο ότι οι αναλογίες θα πληρούνται. Αν δεν πληρούνται, περιμένει είτε μέχρι να φύγει κάποιο άλλο παιδί, είτε μέχρι να μπει ένας νέος δάσκαλος και έτσι να δημιουργηθούν περισσότερες θέσεις για παιδιά. Αντίστοιχα, όταν ένας δάσκαλος επιχειρεί να βγει, πρέπει να είμαστε σίγουροι πως ακόμη και αν φύγει θα υπάρχουν εντός του νηπιαγωγείου αρκετοί δάσκαλοι για τον αριθμό των παιδιών που είναι μέσα. Αν δε γίνεται να φύγει, είτε περιμένει έως ότου να εισέλθει ένας άλλος δάσκαλος είτε αναμένει έως ότου φύγει ένας αριθμός παιδιών ώστε να μπορεί να εξέλθει και αυτός χωρίς να υπάρχει πρόβλημα.
Ο κώδικας που υλοποιεί τα παραπάνω είναι ο ακόλουθος:
(χρησιμοποιήσαμε 2 μεταβλητές κατάστασης, 1 για τα παιδιά και 1 για τους δασκάλους)

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

/*
 * POSIX thread functions do not return error numbers in errno,
 * but in the actual return value of the function call instead.
```

```
 * This macro helps with error reporting in this case.
 */
#define perror_pthread(ret, msg) \
	do { errno = ret; perror(msg); } while (0)

/* A virtual kindergarten */
struct kgarten_struct {

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

	/* ... */

	/*
	 * You may NOT modify or use anything in the structure below this
	 * point. They are only meant to be used by the framework code,
	 * for verification.
	 */

	pthread_cond_t cond1,cond2;
	//pthread_cont_t cond2;
	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);

        while((thr->kg->vc) >= ( (thr->kg->vt) * (thr->kg->ratio) )){
            pthread_cond_wait(&thr->kg->cond1, &thr->kg->mutex);
        }

        ++(thr->kg->vc);
        pthread_mutex_unlock(&thr->kg->mutex);
}

void child_exit(struct thread_info_struct *thr)
{

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

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

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

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

	pthread_cond_broadcast(&thr->kg->cond1); //broadcast signal for children enter

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

	 if ( thr->kg->vc >= ((thr->kg->vt -1 ) *( thr->kg->ratio)))
		pthread_cond_broadcast(&thr->kg->cond2); //broadcast signal for teachers exit

	pthread_mutex_unlock(&thr->kg->mutex);
}

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

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

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

	pthread_cond_broadcast(&thr->kg->cond1); //broadcast signal for children enter

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

	pthread_cond_broadcast(&thr->kg->cond2); //broadcast signal for teachers exit

	pthread_mutex_unlock(&thr->kg->mutex);
}

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

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

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

	while ( thr->kg->vc >= ((thr->kg->vt -1 ) *( thr->kg->ratio))){
		pthread_cond_wait(&thr->kg->cond2, &thr->kg->mutex);
	}

	--(thr->kg->vt);
	pthread_mutex_unlock(&thr->kg->mutex);
```

```
}

/*
 * Verify the state of the kindergarten.
 */
void verify(struct thread_info_struct *thr)
{
	struct kgarten_struct *kg = thr->kg;
	int t, c, r;

	c = kg->vc;
	t = kg->vt;
	r = kg->ratio;

	fprintf(stderr, "	Thread %d: Teachers: %d, Children: %d\n",
		thr->thrid, t, c);

	if (c > t * r) {
		bad_thing(thr->thrid, c, t);
		exit(1);
	}
}


/*
 * A single thread.
 * It simulates either a teacher, or a child.
 */
void *thread_start_fn(void *arg)
{
	/* We know arg points to an instance of thread_info_struct */
	struct thread_info_struct *thr = arg;
	char *nstr;

	fprintf(stderr, "Thread %d of %d. START.\n", thr->thrid, thr->thrcnt);

	nstr = thr->is_child ? "Child" : "Teacher";
	for (;;) {
		fprintf(stderr, "Thread %d [%s]: Entering.\n", thr->thrid, nstr);
		if (thr->is_child)
			child_enter(thr);
		else
			teacher_enter(thr);

		fprintf(stderr, "Thread %d [%s]: Entered.\n", thr->thrid, nstr);

		/*
		 * We're inside the critical section,
		 * just sleep for a while.
		 */
		/* usleep(rand_r(&thr->rseed) % 1000000 / (thr->is_child ? 10000 : 1)); */
		pthread_mutex_lock(&thr->kg->mutex);
```

```
            verify(thr);
            pthread_mutex_unlock(&thr->kg->mutex);

            usleep(rand_r(&thr->rseed) % 1000000);

            fprintf(stderr, "Thread %d [%s]: Exiting.\n", thr->thrid, nstr);
            /* CRITICAL SECTION END */

            if (thr->is_child)
                  child_exit(thr);
            else
                  teacher_exit(thr);

            fprintf(stderr, "Thread %d [%s]: Exited.\n", thr->thrid, nstr);

            /* Sleep for a while before re-entering */
            /* usleep(rand_r(&thr->rseed) % 100000 * (thr->is_child ? 100 : 1)); */
            usleep(rand_r(&thr->rseed) % 100000);

            pthread_mutex_lock(&thr→kg→mutex);
            verify(thr);
            pthread_mutex_unlock(&thr->kg->mutex);
      }

      fprintf(stderr, "Thread %d of %d. END.\n", thr->thrid, thr->thrcnt);

      return NULL;
}


int main(int argc, char *argv[])
{
      int i, ret, thrcnt, chldcnt, ratio;
      struct thread_info_struct *thr;
      struct kgarten_struct *kg;

      /*
       * Parse the command line
       */
      if (argc != 4)
            usage(argv[0]);
      if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0) {
            fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
            exit(1);
      }
      if (safe_atoi(argv[2], &chldcnt) < 0 || chldcnt < 0 || chldcnt > thrcnt) {
            fprintf(stderr, "`%s' is not valid for `child_threads'\n", argv[2]);
            exit(1);
      }
      if (safe_atoi(argv[3], &ratio) < 0 || ratio < 1) {
            fprintf(stderr, "`%s' is not valid for `c_t_ratio'\n", argv[3]);
            exit(1);
```

```
	}


	/*
	 * Initialize kindergarten and random number generator
	 */
	srand(time(NULL));
kg = safe_malloc(sizeof(*kg));
	kg->vt = kg->vc = 0;
	kg->ratio = ratio;


	ret = pthread_mutex_init(&kg->mutex, NULL);
	if (ret) {
		perror_pthread(ret, "pthread_mutex_init");
		exit(1);
	}

	// arxikopoihsh condvar
	ret = pthread_cond_init(&kg->cond1,NULL);
	if(ret){
		perror_pthread(ret, "pthread_cond_init");
	}

	ret = pthread_cond_init(&kg->cond2,NULL);
	if(ret){
		perror_pthread(ret, "pthread_cond_init");
	}

	/*
	 * Create threads
	 */

	thr = safe_malloc(thrcnt * sizeof(*thr));

	for (i = 0; i < thrcnt; i++) {
		/* Initialize per-thread structure */
		thr[i].kg = kg;
		thr[i].thrid = i;
		thr[i].thrcnt = thrcnt;
		thr[i].is_child = (i < chldcnt);
		thr[i].rseed = rand();

		/* Spawn new thread */
		ret = pthread_create(&thr[i].tid, NULL, thread_start_fn, &thr[i]);
		if (ret) {
			perror_pthread(ret, "pthread_create");
			exit(1);
		}
	}

	/*
```

```
	 * Wait for all threads to terminate
	 */
	for (i = 0; i < thrcnt; i++) {
		ret = pthread_join(thr[i].tid, NULL);
		if (ret) {
			perror_pthread(ret, "pthread_join");
			exit(1);
		}
	}

	printf("OK.\n");

	return 0;
}
```

Το πρόγραμμα που παρατέθηκε παραπάνω είναι συνεχούς λειτουργίας.
Αν κληθέι με παραμέτρους 10(άτομα συνολικά) 7(παιδιά) 2(αναλογία για κάθε καθηγητή) μας δίνει:

```
Thread 1 of 10. START.
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 2 of 10. START.
Thread 2 [Child]: Entering.
THREAD 2: CHILD ENTER
Thread 3 of 10. START.
Thread 3 [Child]: Entering.
THREAD 3: CHILD ENTER
Thread 4 of 10. START.
Thread 4 [Child]: Entering.
THREAD 4: CHILD ENTER
Thread 5 of 10. START.
Thread 5 [Child]: Entering.
THREAD 5: CHILD ENTER
Thread 6 of 10. START.
Thread 6 [Child]: Entering.
THREAD 6: CHILD ENTER
Thread 7 of 10. START.
Thread 7 [Teacher]: Entering.
THREAD 7: TEACHER ENTER
Thread 7 [Teacher]: Entered.
            Thread 7: Teachers: 1, Children: 0
Thread 1 [Child]: Entered.
            Thread 1: Teachers: 1, Children: 1
Thread 2 [Child]: Entered.
            Thread 2: Teachers: 1, Children: 2
Thread 8 of 10. START.
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
Thread 8 [Teacher]: Entered.
            Thread 8: Teachers: 2, Children: 2
Thread 3 [Child]: Entered.
            Thread 3: Teachers: 2, Children: 3
Thread 4 [Child]: Entered.
            Thread 4: Teachers: 2, Children: 4
Thread 9 of 10. START.
```

```
Thread 9 [Teacher]: Entering.
THREAD 9: TEACHER ENTER
Thread 9 [Teacher]: Entered.
            Thread 9: Teachers: 3, Children: 4
Thread 5 [Child]: Entered.
            Thread 5: Teachers: 3, Children: 5
Thread 6 [Child]: Entered.
            Thread 6: Teachers: 3, Children: 6
Thread 0 of 10. START.
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
Thread 4 [Child]: Exiting.
THREAD 4: CHILD EXIT
Thread 4 [Child]: Exited.
Thread 0 [Child]: Entered.
            Thread 0: Teachers: 3, Children: 6
Thread 5 [Child]: Exiting.
THREAD 5: CHILD EXIT
Thread 5 [Child]: Exited.
            Thread 5: Teachers: 3, Children: 5
Thread 5 [Child]: Entering.
THREAD 5: CHILD ENTER
Thread 5 [Child]: Entered.
            Thread 5: Teachers: 3, Children: 6
            Thread 4: Teachers: 3, Children: 6
Thread 4 [Child]: Entering.
THREAD 4: CHILD ENTER
Thread 0 [Child]: Exiting.
THREAD 0: CHILD EXIT
Thread 0 [Child]: Exited.
Thread 4 [Child]: Entered.
            Thread 4: Teachers: 3, Children: 6
Thread 7 [Teacher]: Exiting.
THREAD 7: TEACHER EXIT
            Thread 0: Teachers: 3, Children: 6
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
Thread 2 [Child]: Exiting.
THREAD 2: CHILD EXIT
Thread 2 [Child]: Exited.
```

1. Έστω ότι ένας από τους δασκάλους έχει αποφασίσει να φύγει, αλλά δε 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;

Αν ένας από τους δασκάλους έχει αποφασίσει να φύγει, είναι πιθανό να μην τα καταφέρει στην περίπτωση που λόγω της απουσίας του πάψει να τηρείται η ζητούμενη αναλογία δασκάλων-παιδιών. Τότε πρέπει να περιμένει μέχρι είτε να μειωθεί ο αριθμός των παιδιών στο χώρο του νηπιαγωγείου, είτε μέχρι να εισέλθει άλλος δάσκαλος (σε εκείνες τις περιπτώσεις στέλνουμε σήμα στην cond2) . Βέβαια, αν ένα παιδί καταφτάσει και επιχειρήσει να μπει στο χώρο, αν η αναλογία τηρεί τις προδιαγραφές, είναι πιθανό να καταφέρει να μπει παρόλο που ο δάσκαλος περιμένει ήδη να βγει. Αυτό συμβαίνει παρόλο που χρησιμοποιούμε διαφορετικές condition variables για δασκάλους και παιδιά, αφού στέλνουμε σήμα και στους 2 ταυτόχρονα χωρίς να δίνουμε συγκεκριμένη προτεραιότητα για το ποια από τις διεργασίες που βρίσκονται σε αναμονή θα “ξυπνήσει” και θα μεταβεί στο κρίσιμο τμήμα της. Επομένως το ποια διεργασία θα τρέξει είναι τυχαίο και όχι ντετερμινιστικό στην περίπτωσή μας. Αυτό που είναι βέβαιο πάντως είναι ότι μόνο ένα αίτημα θα εξυπηρετηθεί ώστε να μην οδηγηθούμε σε καμία περίπτωση σε απαγορευμένη κατάσταση για το νηπιαγωγείο.

2.Υπάρχουν καταστάσεις συναγωνισμού (races) στον κώδικα kgarten.c που επιχειρεί να επαληθεύσει την ορθότητα του σχήματος συγχρονισμού που υλοποιείτε; Αν όχι , εξηγείστε γιατί. Αν ναι, δώστε παράδειγμα μιας τέτοιας κατάστασης.

Στο kgarten.c ανακύπτουν race conditions τα οποία επιλύονται από το ίδιο το πρόγραμμα. Ειδικότερα, για τον έλεγχο των τηρούμενων αναλογιών δάσκαλων-μαθητών γίνεται κλήση της συνάρτησης verify, η οποία υπολογίζει αυτή την αναλογία. Αν, τη στιγμή που ένα thread καλούσε την verify ένα δεύτερο block τροποποιούσε την τιμή κάποιας από τις μεταβλητές vt-vc που δηλώνουν τον αριθμό καθηγητών-παιδιών αντίστοιχα ή καλούσε ξανά την verify, τότε η συνάρτηση θα επέστρεφε λάθος αποτέλεσμα.Επομένως γι'αυτό χρησιμοποιείται mutex, κλειδώνει πριν την κληση της verify από ένα thread αλλά και κατά την αύξηση ή μείωση των μεταβλητών vt-vc και ξεκλειδώνει μετά. Έτσι, αν ο mutex έχει κλειδώσει, το επόμενο thread που θα δοκιμάσει να τροποποιήσει κάποια από τις μεταβλητές, θα περιμένει μέχρι να βγει το thread από το κρίσιμο τμήμα και να ξεκλειδώσει ο mutex . Συνολικά,με τη χρήση mutex διασφαλίζεται ότι όταν ένα thread βρίσκεται εντός του κρίσιμου τμήματος κανένα άλλο δεν θα εμπλακεί πριν ο mutex ξεκλειδώσει γεγονός που εξασφαλίζει την εγκυρότητα των παραγόμενων εξόδων του προγράμματος.

Επίσης, ο συγχρονισμός προκαλεί καταστάσεις συναγωνισμού όταν κάποιο παιδί θέλει να μπει και όταν κάποιος δάσκαλος θέλει να βγει από υο νηπιαγωγείο. Για να εξασφαλιστεί η τήρηση της ζητούμενης αναλογίας γίνεται χρήση της condition variable cond1 από τα threads που συμβολίζουν τα παιδιά και cond2 από τα threads που συμβολίζουν τους δασκάλους. Έτσι, στην περίπτωση που ένα παιδί ζητήσει να μπει ή ένας δάσκαλος επιχειρήσει να βγει και δεν ισχύει η αναλογία,τότε με την εντολή

pthread_cond_wait(&(thr→kg→cond1),&(thr→kg→mutex)); ή
pthread_cond_wait(&(thr→kg→cond2),&(thr→kg→mutex)); (αντίστοιχα)

ο mutex θα "ξεκλειδώσει" για να χρησιμοποιηθεί από άλλο thread ενώ το συγκεκριμένο thread θα μεταβεί σε κατάσταση waiting. Αν κάποιο παιδί βγει ή ένας δάσκαλος μπει στο νηπιαγωγείο, τότε με την εντολή

pthread_cond_broadcast(&thr→kg→cond1); (θα ξυπνήσει τα παιδιά που θέλουν να μπουν)
pthread_cond_broadcast(&thr→kg→cond2); (θα ξυπνησει τους δασκάλους που θέλουν να βγουν)

θα "ξυπνήσουν" όλα τα threads που είχαν τεθεί σε κατάσταση waiting. Επειδή όμως υπάρχει ένα μόνο mutex, το thread που θα εισέρθει πρώτο στο κρίσιμο τμήμα θα κλειδώσει το mutex και τα υπόλοιπα θα τεθούν ξανά σε κατάσταση waiting -χωρίς να γνωρίζουμε τι θα γίνει ντετερμινιστικά κάθε φορά,δηλαδή ποιο θα εισέλθει πρώτο-.Όταν ξυπνάει ένα thread, συνεχίζει την εκτέλεση μέσα στο σώμα της εντολής while στην οποία ελέγχουμε την αναλογία,απαγορεύοντας του να εισέλθει σε κρίσιμο τμήμα αν δεν πληροί την κατάλληλη συνθήκη.

Τελικά, σε κάθε περίπτωση που ανακύπτουν καταστάσεις ανταγωνισμού διαπιστώνουμε και εξασφαλίζουμε με την λειτουργία του προγράμματος μας ότι δεν θα προκύψουν εσφαλμένα αποτελέσματα.