

Άσκηση 3

Συγχρονισμός

Λειτουργικά Συστήματα

Εργαστηρική Ομάδα B02

Βαβουλιώτης Γεώργιος
A.M. : 03112083
7ο εξάμηνο

Γιαννούλας Βασίλειος
A.M. : 03112117
7ο εξάμηνο

1. Συγχρονισμός σε υπάρχοντα κώδικα

Σκοπός της πρώτης άσκησης είναι ο συγχρονισμός δυο νημάτων τα οποία τρέχουν ταυτόχρονα σε υπάρχον κώδικα . Συγκεκριμένα στο πηγαίο αρχείο με όνομα `simplesync.c` το οποίο μας δίνετε, τα δυο νήματα δρουν πάνω σε μια κοινή μεταβλητή με όνομα `val` (το ένα νήμα αυξάνει την τιμή του `val` κατά 1 N το πλήθος φορές και το άλλο μειώνει τη τιμή του `val` κατά 1 N το πλήθος φορές). Αυτό που θέλουμε να πετύχουμε με τον συγχρονισμό των δυο νημάτων είναι να μείνει ανεπηρέαστη η αρχική τιμή της μεταβλητής `val`, δηλαδή το `val` να είναι ίσο με μηδέν όταν τελειώσει η εκτέλεση του προγράμματος. Θα πρέπει να τονιστεί πως πριν γίνει κάποια αλλαγή-προσθήκη στο δοθέν πρόγραμμα, δηλαδή πριν υλοποιηθεί ο συγχρονισμός των νημάτων η τιμή της μεταβλητής `val` μετά την εκτέλεση του προγράμματος δεν είναι η αναμενόμενη (δηλαδή μηδέν) αφού το ένα νήμα παρεμβάλεται στην εκτέλεση του άλλου.

Ο συγχρονισμός των δυο νημάτων γίνεται με 2 διαφορετικούς τρόπους, ο πρώτος είναι με χρήση ατομικών λειτουργιών του GCC και ο άλλος είναι με χρήση POSIX mutexes. Αρχικά όταν πήγαμε να μεταγλωττίσουμε το δοσμένο κώδικα με χρήση του Makefile που μας δίνετε γράφοντας την εντολή `make` στο terminal, παρατηρούμε ότι έχουν παραχθεί 2 εκτελέσιμα αρχεία με ονόματα `simplesync-atomic` και `simplesync-mutex` (αυτό το βλέπουμε πατώντας `ls`). Το πρώτο αντιστοιχεί στην υλοποίηση του συγχρονισμού με ατομικές λειτουργίες και το δεύτερο με χρήση POSIX mutexes. Η παραγωγή των 2 εκτελέσιμων αρχείων οφείλεται στα εξής :

- Στο δοσμένο αρχείο υπάρχει το παρακάτω κομμάτι κώδικα το οποίο ορίζει ποιό από τα παραπάνω εκτελέσιμα αρχεία θα παραχθεί :

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

- Στο δοσμένο Makefile υπάρχουν οι παρακάτω εντολές, οι οποίες είναι αυτές που ορίζουν ποιό εκτελέσιμο θα παραχθεί :

```
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
```

Τελικά ανάλογα με την τιμή του `USE_ATOMIC_OPS` γίνεται διαφορετικός συγχρονισμός και παράγεται το ανάλογο εκτελέσιμο αρχείο.

Συγχρονισμός με Ατομικές Λειτουργίες του GCC:

Για την υλοποίηση του συγχρονισμού με ατομικές λειτουργίες του GCC χρησιμοποιούνται οι δυο επόμενες εντολές: `__sync_add_and_fetch(ip,1)` και `__sync_sub_and_fetch(ip,1)`. Οι εντολές αυτές, αυτό που κάνουν είναι να δρουν σε επίπεδο υλικού δηλαδή το κλείδωμα το οποίο εξασφαλίζει το συγχρονισμό των νημάτων της άσκησης υλοποιείται στο υλικό. Πιο αναλυτικά, εξασφαλίζουν ότι αρχικά θα αποθηκευτεί η τιμή της μεταβλητής στη θέση μνήμης που της αντιστοιχεί μετά την εφαρμογή της επιθυμητής λειτουργίας και μετά από αυτό θα είναι δυνατή οποιαδήποτε τροποποίηση αυτής, δηλαδή υλοποιείται κλείδωμα σε επίπεδο υλικού. Η σωστή υλοποίηση του συγχρονισμού με ατομικές λειτουργίες φαίνεται και από την έξοδο του εκτελέσιμου `simplesync-atomic`, η οποία φαίνεται παρακάτω:

```
[oslabd02@kefalonia:~/Exercise3$ ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Συγχρονισμός με POSIX mutexes:

Η υλοποίηση του συγχρονισμού με POSIX mutexes επιτυγχάνεται με χρήση κλειδωμάτων. Συγκεκριμένα όταν ένα νήμα θέλει να μπει στο κρίσιμο τμήμα, δηλαδή στο τμήμα στο οποίο πρέπει να βρίσκεται μόνο ένα νήμα κάθε φορά (το τμήμα αυτό είναι εκείνο της αύξησης και της μείωσης της τιμής του `val`), αυτό που κάνει είναι να κλειδώσει ώστε να μην μπορεί να μπει το άλλο νήμα στο τμήμα αυτό και στη συνέχεια όταν κάνει την επιθυμητή λειτουργία ξεκλειδώνει ώστε να μπορεί και το άλλο να εισέλθει στο κρίσιμο τμήμα για να κάνει την δική του λειτουργία. Και με αυτό το τρόπο συγχρονισμού θα καταφέρουμε η τιμή του `val` να παραμείνει 0 στο τέλος του προγράμματος, γεγονός το οποίο φαίνεται και από την έξοδο του εκτελέσιμου `simplesync-mutex`, η οποία φαίνεται παρακάτω :

```
[oslabd02@kefalonia:~/Exercise3$ ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Ο κώδικας του προγράμματος μετά τις απαραίτητες προσθήκες(οι προσθήκες είναι με μπλε χρώμα) που έγιναν για να επιτευχθεί ο συγχρονισμός με τους ζητούμενους τρόπους είναι ο παρακάτω:

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

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

#define N 10000000

pthread_mutex_t mrlock;

#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

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(&mrlock);
            ++(*ip);
            pthread_mutex_unlock(&mrlock);
        }
    }
    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);
        }
    }
}
```

```

    } else {
        pthread_mutex_lock(&mrlock);
        --(*ip);
        pthread_mutex_unlock(&mrlock);
    }
}
fprintf(stderr, "Done decreasing variable.\n");
return NULL;
}

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

    val = 0;

    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);
    }

    ret = pthread_join(t1, NULL);
    if (ret)
        perror_pthread(ret, "pthread_join");
    ret = pthread_join(t2, NULL);
    if (ret)
        perror_pthread(ret, "pthread_join");

    ok = (val == 0);

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

    return ok;
}

```

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;

Το παρακάτω πίνακάκι περιλαμβάνει όλους τους χρόνους για καθένα από τα δυο εκτελέσιμα που έχουμε με και χωρίς συγχρονισμό (δεν παρέθεσα screenshot αφού θα ήταν 4 εικόνες συνολικά ενώ με τον πίνακα είναι πιο σύντομο):

	simplesync-atomic	simplesync-mutex
Χωρίς Συγχρονισμό	0,036 sec	0,052 sec
Με Συγχρονισμό	0,892 sec	1,720 sec

Σχολιασμός: Από τον παραπάνω πίνακα είναι προφανές ότι αν εφαρμόσουμε συγχρονισμό (με οποιοδήποτε τρόπο) το πρόγραμμά μας χρειάζεται περισσότερο χρόνο για να εκτελεστεί συγκριτικά με το χρόνο που θα χρειαζόταν αν δεν εφαρμόζαμε συγχρονισμό. Το γεγονός αυτό είναι απολύτως λογικό διότι για να κάνει μια ενέργεια ένα νήμα θα πρέπει πριν το άλλο να έχει τελειώσει άρα το ένα νήμα περιμένει το άλλο, πράγμα το οποίο δικαιολογεί απόλυτα την αύξηση του χρόνου εκτέλεσης του προγράμματος αν εφαρμόσουμε συγχρονισμό.

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

Αν κοιτάξουμε εκ νέου το πίνακάκι το οποίο παρατέθηκε στην προηγούμενη ερώτηση παρατηρούμε πως ο συγχρονισμός με χρήση ατομικών λειτουργιών του GCC είναι κατά πολύ γρηγορότερος από αυτόν που χρησιμοποιεί POSIX mutexes. Κάτι τέτοιο είναι λογικό διότι καθένα από τα 2 εκτελέσιμα που δημιουργούνται έχει διαφορετικό κώδικα assembly. Συγκεκριμένα, όταν κάναμε χρήση ατομικών λειτουργιών χρησιμοποιήσαμε τις εντολές `__sync_add_and_fetch(ip,1)` και `__sync_snb_and_fetch(ip,1)` για να κάνουμε τον συγχρονισμό και την αύξηση-μείωση του `val`, οι οποίες μεταφράζονται σε 1 μόνο εντολή assembly, ενώ όταν χρησιμοποιήσαμε POSIX mutexes το τμήμα του κώδικα που κάνει το ζητούμενο συγχρονισμό αντιστοιχεί σε παραπάνω εντολές assembly. Επομένως, όταν ένα νήμα μπαίνει στο κρίσιμο τμήμα χρειάζεται λιγότερους κύκλους μηχανής για να κλειδώσει, να ολοκληρώσει την ενέργεια του και να ξεκλειδώσει όταν χρησιμοποιούμε ατομικές λειτουργίες GCC σε σχέση με τα POSIX mutexes. Επειδή λοιπόν κάθε νήμα μπαίνει στο κρίσιμο τμήμα πολλές φορές έχω πολύ περισσότερους κύκλους μηχανής στην περίπτωση των mutexes άρα και μεγαλύτερο χρόνο εκτέλεσης, το οποίο επιβεβαιώνεται και από τα αποτελέσματα που έχει το πίνακάκι.

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.

Η εντολή `__sync_add_and_fetch(ip,1)` μεταφράζεται σύμφωνα με τα παραπάνω στην εξής εντολή assembly: `lock addl $1, (%ebx)`

Η εντολή `__sync_sub_and_fetch(ip,1)` μεταφράζεται σύμφωνα με τα παραπάνω στην εξής εντολή assembly: `lock addl $-1, (%ebx)`

Παρατηρώ ότι όντως οι ατομικές εντολές μεταφράζονται σε μία μόνο εντολή assembly, όπως δηλαδή ανέφερα στην προηγούμενη ερώτηση.

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.

Όσο αφορά τα POSIX mutexes προέκυψε ότι:

- η εντολή κλειδώματος μεταφράζεται στις παρακάτω εντολές assembly :
`movl $mutex1, (%esp)`
`call pthread_mutex_lock`
- η εντολή ξεκλειδώματος μεταφράζεται στις παρακάτω εντολές assembly :
`movl $mutex1, (%esp)`
`call pthread_mutex_unlock`

Γενικό Σχόλιο: Απο τις ερωτήσεις 3 και 4 επιβεβαιώθηκε και πρακτικά πως ο συγχρονισμός με χρήση ατομικών λειτουργιών του GCC απαιτεί λιγότερους κύκλους μηχανής(λιγότερες εντολές assembly) σε σχέση με την χρήση POSIX mutexes, δηλαδή είναι γρηγορότερη μέθοδος συγχρονισμού.

2. Παράλληλος υπολογισμός του συνόλου Mandelbrot

Στην άσκηση αυτή μας δίνεται το πηγαίο αρχείο με όνομα `mandel.c` το οποίο περιέχει τον κώδικα που υπολογίζει και σχεδιάζει την ακολουθία του συνόλου Mandelbrot. Στόχος αυτής της άσκησης είναι η κατανομή της διαδικασίας υπολογισμού και σχεδιασμού του συνόλου Mandelbrot σε κάποια νήματα(το πόσα είναι δίνετε σαν όρισμα όταν τρέχω το εκτελέσιμο). Ωστόσο για να είναι κάτι τέτοιο εφικτό θα πρέπει να τροποποιήσουμε τον δοσμένο κώδικα ώστε να χρησιμοποιεί σχήμα συγχρονισμού έτσι ώστε ένα νήμα να μην παρεμβαίνει στην ενέργεια κάποιου άλλου, διότι αν το επιτρέψουμε αυτό δεν θα πάρουμε το σωστό αποτέλεσμα.

Στον κώδικα που σας παραθέτω παρακάτω υλοποιήθηκε η εξής ιδέα : Αρχικά δημιουργώ ένα πίνακα από νήματα(το πόσα είναι τα νήματα δίνεται σαν όρισμα) του οποίου κάθε κελί είναι ένα struct με τις πληροφορίες του συγκεκριμένου νήματος. Επίσης για να επιτύχω τον συγχρονισμό χρησιμοποιώ ένα πίνακα σημαφόρων, όπου το πλήθος το σημαφόρων ισούται με το πλήθος των νημάτων. Επομένως υπάρχει μια αντιστοιχία 1-1 ανάμεσα στα νήματα και στους σημαφόρους με την έννοια ότι το νήμα με αριθμό *i* έχει ως σημαφόρο τον σημαφόρο *i* και μόνο αυτόν. Αυτό που πρέπει να κάνω αρχικά είναι να αρχικοποιήσω όλους τους σημαφόρους στο 0 με χρήση των εντολών : `sem_init(&semaphore[line], 0, 1);`
`sem_wait(&semaphore[line]);` (θα μπορούσα να βάλω μόνο την εντολή `sem_init(&semaphore[line]), 0, 0)`). Κάθε σημαφόρος είναι αρχικοποιημένος στο 0 και κάθε νήμα βρίσκεται στη συνάρτηση(`thread_start_fn`) που δίνεται σαν όρισμα στην `pthread_create(&tarray[line].tid, NULL, thread_start_fn, &tarray[line])` και εκτελεί τον κώδικά της, υπολογίζοντας κάθετα την γραμμή που του αντιστοιχεί. Όταν κάθε νήμα συναντήσει την εντολή `sem_wait(&semaphore[line%thr->numofthrd]);` τότε επειδή έχω αρχικοποιήσει τους σημαφόρους στο 0 μπλοκάρει η εκτέλεση του νήματος έως ότου σταλθεί σε αυτό σήμα εκκίνησης με την `sem_post()`, δηλαδή αύξηση του αντίστοιχου σεμαφόρου. Άρα τώρα είμαι στην κατάσταση στην οποία όλα τα νήματα έχουν μπλοκάρει και περιμένουν να τους έρθει κάποιο `sem_post()`. Στο σημείο αυτό στέλνω `sem_post()` στο σεμαφόρο που αντιστοιχεί στο νήμα 0 (τα νήματα αριθμούνται από 0 έως `Nthreads-1`) με την εντολή `sem_post(&semaphore[0]);` και πλέον μόνο το νήμα 0 φεύγει από blocking και εκτελεί την επόμενη εντολή. Μόλις εμφανίσει στην έξοδο την γραμμή που του έχει ανατεθεί στέλνει `sem_post()` στο επόμενο νήμα(δηλαδή στο νήμα 1 αρχικά) για να τυπώσει κι εκείνο την γραμμή του. Όταν πλέον έχω τυπώσει μέχρι και την `NTHREADS` γραμμή, δηλαδή κάθε νήμα έχει τυπώσει από μια γραμμή εξασφαλίζω ότι το σήμα `sem_post()` θα σταλεί στο νήμα 0 ξανά με χρήση του τελεστή `%(modulo)` και θα επαναληφθεί η ίδια διαδικασία μέχρι να τυπωθούν στην έξοδο όλες οι γραμμές. Επομένως οι γραμμές εμφανίζονται μία μία ενώ υπολογίζονται παράλληλα χωρίς να δημιουργείται πρόβλημα στο τελικό αποτέλεσμα.

Παρατήρηση: Όπου χρησιμοποιώ το `sem_post()` χωρίς να βάλω μέσα παράμετρο δεν σημαίνει πως δεν πρέπει να έχει παράμετρο απλά εγώ την αμελώ για να εξηγήσω όσο πιο απλά γίνεται την ιδέα της άσκησης.

Ο κώδικας όπως αυτός έγινε μετά τις απαιτούμενες προσθήκες και τροποποιήσεις ώστε να υλοποιηθεί το ζητούμενο φαίνεται παρακάτω (τα σημεία του κώδικα με **κόκκινο χρώμα** είναι αυτά τα οποία **πρόσθεσα εγώ και δεν ήταν έτοιμα** ώστε να πετύχω συγχρονισμό, αλλά προφανώς πρόσθεσα κι αλλά πράγματα(πχ πίνακα για νήματα κλπ) τα οποία όμως δεν τα επισημαίνω με κάποιο χρώμα) :


```

#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)

int y_chars = 50;
int x_chars = 90;

sem_t *semaphore;

double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

double xstep;
double ystep;

struct thread_info_struct {
    pthread_t tid; /* POSIX thread id, as returned by the library */
    int tnumber;
    int numofthrd;
};
typedef struct thread_info_struct * threadptr;

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 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;
    }
}

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 */
    threadptr thr = arg;

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

    for (line = thr->tnumber ; line < y_chars; line += thr->numofthrd ) {
        compute_mandel_line(line, color_val);
        sem_wait(&semaphore[line%thr->numofthrd]);
        output_mandel_line(1, color_val);
        sem_post(&semaphore[(thr->tnumber +1)%(thr->numofthrd)]);
    }

    return NULL;
}

int main(int argc, char *argv[])
{
    int line, NTHREADS, i;
    threadptr tarray;
    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);
    }

    tarray = safe_malloc(NTHREADS * sizeof(*tarray));
    semaphore = safe_malloc(NTHREADS * sizeof(*semaphore));

    for(line = 0 ; line < NTHREADS ; line++){
        sem_init(&semaphore[line], 0, 1);
        sem_wait(&semaphore[line]);
    }

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

    sem_post(&semaphore[0]);
    for (i = 0; i < NTHREADS; i++) {

```

```

ret = pthread_join(tarray[i].tid, NULL);
if (ret) {
    perror_pthread(ret, "pthread_join");
    exit(1);
}
}

reset_xterm_color(1);
return 0;
}

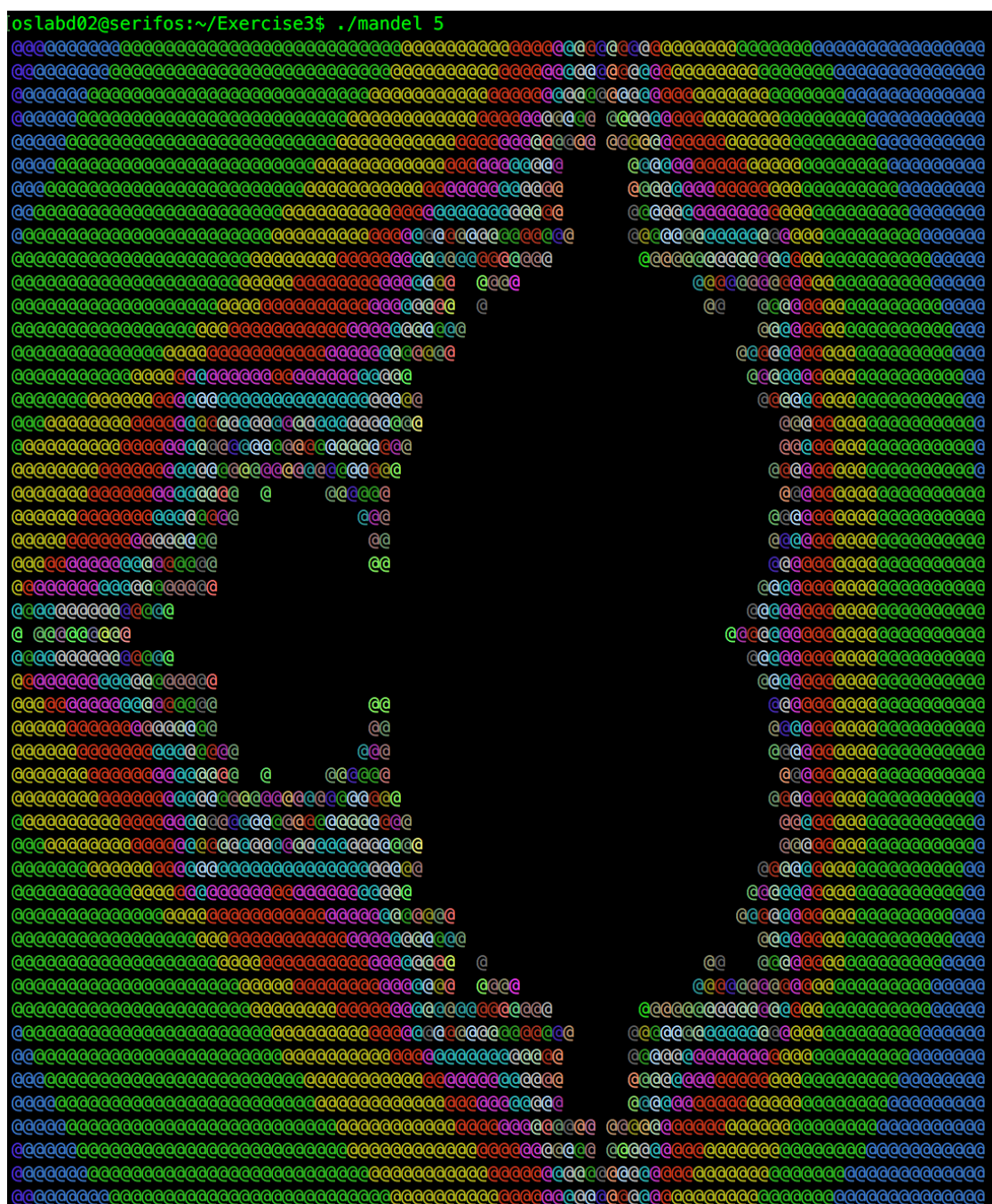
```

Αφού μεταγλωττίσω το τροποποιημένο αρχείο mandel.c με την βοήθεια του Makefile που μου δίνετε, μπορώ να τρέξω το εκτελέσιμο αρχείο που παράγεται, δίνοντας του σαν παράμετρο τον αριθμό των νημάτων. Ανεξάρτητα με την παράμετρο που θα του δώσω, η έξοδος που θα πάρω είναι η εξής(εδώ έδωσα ως παράμετρο το 5):

```

oslabd02@serifos:~/Exercise3$ ./mandel 5

```



Ερωτήσεις

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

Στην άσκηση αυτή χρησιμοποιώ ένα πίνακα απο σημαφόρους για να επιτύχω τον συγχρονισμό των νημάτων, ο οποίος είναι μεγέθους όσο και το πλήθος των νημάτων, δηλαδή υπάρχει μια 1-1 αντιστοιχία μεταξύ νημάτων και σημαφόρων όπως ήδη έχω αναφέρει στην εξήγηση της ιδέας που υλοποίησα. Όταν στείλω `sem_post()` στο πρώτο νήμα θα τυπώσει την γραμμή του και στην συνέχεια θα στείλει με την σειρά του `sem_post()` στο επόμενο νήμα και θα ακολουθηθεί αυτή η κυκλική αποστολή `sem_post()` μέχρι να τυπωθούν όλες οι γραμμές. Αναφέρω εκ νέου ότι αν αριθμός των νημάτων είναι μικρότερος από τον αριθμό των γραμμών που έχω να εκτυπώσω τότε η `NTHREADS+1` γραμμής υπολογίζεται και εκτυπώνεται απο το νήμα 0 κ.ο.κ.

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.

Σειριακός Υπολογισμός : Στην περίπτωση του σειριακού υπολογισμού του συνόλου έχω τους εξής χρόνους :

```
real      0m1.348s
user      0m1.240s
sys       0m0.016s
oslabd02@lemnos:~/Exercise3$
```

Παράλληλος Υπολογισμός : Στην περίπτωση του παράλληλου υπολογισμού του συνόλου με χρήση 2 νημάτων έχω τους εξής χρόνους:

```
real      0m0.664s
user      0m1.248s
sys       0m0.004s
oslabd02@lemnos:~/Exercise3$
```

- 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;**

Απο τους χρόνους που παρατέθηκαν στην προηγούμενη ερώτηση παρατηρώ ότι υπάρχει επιτάχυνση στο χρόνο εκτέλεσης του προγράμματος όταν αυτό εκτελείται σε διπύρηνο επεξεργαστή. Το γεγονός αυτό είναι απόλυτα λογικό, διότι αν έχω 2 πυρήνες η δουλειά που απαιτείται για να παραχθεί το αποτέλεσμα μοιράζεται σε 2 πόρους, ενώ στη σειριακή εκτέλεση όλη η δουλειά γινόταν απο ενα μόνο πόρο. Επίσης, στο πρόγραμμα μας το κρίσιμο τμήμα, δηλαδή το τμήμα κώδικα στο οποίο θέλουμε να είναι μέσα μόνο ενα νήμα κάθε φορά, είναι αυτό στο οποίο γίνεται η εκτύπωση της κάθε γραμμής, διότι κάθε φορά θέλω μόνο ενα νήμα να εκτυπώνει την γραμμή που έχει υπολογίσει, ώστε να πάρω το σωστό αποτέλεσμα τελικά. Παρατηρούμε λοιπόν πως το κρίσιμο τμήμα είναι μικρό διότι η εκτύπωση μιας γραμμής γίνεται γρήγορα και γι' αυτο το λόγο το μπλοκάρισμα των άλλων νημάτων γίνεται για μικρό χρονικό διάστημα. Επίσης, ο υπολογισμός κάθε γραμμής που είναι μια χρονοβόρα διαδικασία, γίνεται απ' όλα τα νήματα ταυτόχρονα και δεν βρίσκεται μέσα στο κρίσιμο τμήμα, γεγονός το οποίο διακαίολογεί προφανώς την μείωση του χρόνου εκτέλεσης του προγράμματος.

- 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 κι αν ο χρήστης πατήσει Ctrl-C, το τερματικό θα επαναφέρεται στην προηγούμενη κατάσταση του;**

Στην περίπτωση που πατήσω Ctrl-C για κάποιο λόγο ενώ το πρόγραμμα εκτελείται τότε διακόπτεται η λειτουργία του. Στο τερματικό θα είναι τυπωμένο όσο απο το σύνολο Mandelbrot είχε προλάβει να τυπωθεί και πλέον το τερματικό θα αναμένει την εισαγωγή απο τον χρήστη της επόμενης εντολής. Όταν ο χρήστης εισάγει ένα οποιοδήποτε χαρακτήρα θα διαπιστώσει ότι έχει το χρώμα του τελευταίου χαρακτήρα που εμφανίστηκε για το Mandelbrot λόγω του αυτόματου τερματισμού που προκλήθηκε εξαιτίας της εντολής Ctrl-C. Αν εμείς επιθυμούμε να επεκτείνουμε κάτι στο κώδικά μας ώστε ακόμα και αν πατήσουμε Ctrl-C να επανέρχονται τα χρώματα του τερματικού θα πρέπει να κάνουμε το εξής: Ουσιαστικά πατώντας Ctrl-C έγινε μια ξαφνική διακοπή του προγράμματός μας την οποία εμείς μέχρι τώρα δεν είχαμε κάτι για να την διαχειριστούμε. Πατώντας λοιπόν το Ctrl-C στην ουσία στέλνεται ενα σήμα στο πρόγραμμα μας το οποίο είναι σήμα άμεσου τερματισμού. Αν εμείς είχαμε προβλέψει να φτιάξουμε μια ρουτίνα `sighandler` που θα πιάνει το σήμα που στέλνει το Ctrl-C , θα μπορούσαμε να είχαμε βάλει μέσα σ' αυτή την εντολή που καλεί την συνάρτηση που επαναφέρει το χρώμα του τερματικού(`reset_xterm_color(1)`) και μετά θα τερματίζει το πρόγραμμα. Με τον τρόπο αυτό θα καταφέραμε να επαναφέρουμε το χρώμα του τερματικού ακόμα και αν πατήσουμε Ctrl-C.

3. Επίλυση προβλήματος συγχρονισμού

Στην τρίτη άσκηση γίνεται προσομοίωση της λειτουργίας ενός νηπιαγωγείου. Στο νηπιαγωγείο(το οποίο αρχικά είναι άδειο) υπάρχουν παιδιά και δάσκαλοι. Τα παιδιά και οι δάσκαλοι προσομοιώνονται απο νήματα και ζητούμενο είναι να τηρείται κάθε στιγμή η σωστή αναλογία μαθητών-δασκάλων ώστε να λειτουργεί σωστά το νηπιαγωγείο και τα παιδιά να είναι ασφαλή. Θα πρέπει να τονιστεί οτι ένας δάσκαλος μπορεί να μπει όποτε θέλει στο νηπιαγωγείο αφού δεν χαλάει την αναλογία, αλλά επιτρέπει την επιπλέον είσοδο παιδιών μέχρι το ratio να φτάσει το μέγιστό το οποίο δίνετε στην εκφώνηση. Επίσης, ενα παιδί μπορεί κι αυτό όποτε θέλει να βγει απο το νηπιαγωγείο αφού σίγουρα οι δάσκαλοι που είναι μέσα θα επαρκούν για να τηρηθεί η ασφάλεια. Τα σημεία τα οποία οφείλουν να προσεχθούν είναι όταν κάποιο(κάποια) παιδί(παιδιά) θέλει να μπει στο νηπιαγωγείο και όταν ένας δάσκαλος θέλει να βγει. Συγκεκριμένα, στο νηπιαγωγείο μπαίνουν παιδιά μέχρι να γεμίσουν όλες οι θέσεις. Αν τώρα οι θέσεις γεμίσουν και υπάρχουν κι άλλα παιδιά που θέλουν να μπουν θα πρέπει αυτά να περιμένουν εως ότου φύγει κάποιο παιδι ή μπει ενας καινούριος δάσκαλος. Επίσης, αν ένας δάσκαλος θέλει να βγει θα πρέπει να περιμένει εως ότου οι υπόλοιποι δάσκαλοι που είναι στο νηπιαγωγείο επαρκούν ώστε να προσέχουν τα παιδιά που είναι μέσα και μόνο τότε μπορεί να φύγει. Τέλος, τα κρίσιμα σημεία του προγράμματος αυτού είναι εκείνα στα οποία γίνεται η αύξηση και η μείωση του πλήθους των δασκάλων και των παιδιών καθώς και εκεί που γίνεται ο έλεγχος αν το νηπιαγωγείο λειτουργεί σωστά(τον έλεγχο τον κάνει η συνάρτηση `verify()`). Ο έλεγχος των κρίσιμων σημείων γίνεται με χρήση μεταβλητών κατάστασης και κλειδωμάτων, όπως είναι εμφανές και απο τον κώδικα του προγράμματος, ο οποίος (τα σημεία του κώδικα με **κόκκινο χρώμα** είναι αυτά τα οποία **πρόσθεσα εγω και δεν ήταν έτοιμα** ώστε να πετύχω συγχρονισμό) :

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

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

/* A virtual kindergarten */
struct kgarten_struct {
    pthread_cond_t condvar;
    int vt;
    int vc;
    int ratio;
    pthread_mutex_t mutex;
};

struct thread_info_struct {
    pthread_t tid;
    struct kgarten_struct *kg;
    int is_child;
    int thrid;
    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->condvar, &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->condvar);

    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->condvar);

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->condvar, &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;

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

// arxikopoihsh condvar
ret = pthread_cond_init(&kg->condvar, 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;
}

```

Προφανώς το πρόγραμμα kgarten.c είναι ένα πρόγραμμα συνεχούς λειτουργίας άρα δεν μπορούν να δοθούν όλα τα αποτελεσματα του, γι αυτό και στη συνέχεια σας παραθέτω ένα στιγμιότυπο από την εκτέλεση του ./kgarten 10 7 2 :

```
Thread 0: Teachers: 3, Children: 6
Thread 0 [Child]: Entering.
THREAD 0: CHILD ENTER
Thread 5 [Child]: Exiting.
THREAD 5: CHILD EXIT
Thread 5 [Child]: Exited.
Thread 0 [Child]: Entered.
Thread 0: Teachers: 3, Children: 6
Thread 5: Teachers: 3, Children: 6
Thread 5 [Child]: Entering.
THREAD 5: CHILD ENTER
Thread 0 [Child]: Exiting.
THREAD 0: CHILD EXIT
Thread 0 [Child]: Exited.
Thread 5 [Child]: Entered.
Thread 5: Teachers: 3, Children: 6
Thread 0: Teachers: 3, Children: 6
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 4: Teachers: 3, Children: 6
Thread 4 [Child]: Entering.
THREAD 4: CHILD ENTER
Thread 6 [Child]: Exiting.
THREAD 6: CHILD EXIT
Thread 6 [Child]: Exited.
Thread 4 [Child]: Entered.
Thread 4: Teachers: 3, Children: 6
Thread 2 [Child]: Exiting.
THREAD 2: CHILD EXIT
Thread 2 [Child]: Exited.
Thread 1 [Child]: Exiting.
THREAD 1: CHILD EXIT
Thread 1 [Child]: Exited.
Thread 1: Teachers: 3, Children: 4
Thread 1 [Child]: Entering.
THREAD 1: CHILD ENTER
Thread 1 [Child]: Entered.
Thread 1: Teachers: 3, Children: 5
Thread 6: Teachers: 3, Children: 5
Thread 6 [Child]: Entering.
THREAD 6: CHILD ENTER
Thread 6 [Child]: Entered.
Thread 6: Teachers: 3, Children: 6
Thread 2: Teachers: 3, Children: 6
Thread 2 [Child]: Entering.
THREAD 2: CHILD ENTER
```

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 μουν στο χώρο;

Όπως ανέφερα και παραπάνω, όταν ένας δάσκαλος θέλει να φύγει θα πρέπει οι εναπομείναντες δάσκαλοι να μπορούν να επιβλεπουν τα παιδιά που υπάρχουν μέσα στο νηπιαγωγείο, δηλαδή να ισχύει ότι $(\text{Πλήθος Παιδιών}) \leq (\text{Πλήθος Δασκάλων} - 1) * (\text{Ratio})$. Αν ισχύει η σχέση αυτή τότε μπορεί να φύγει ο δάσκαλος, ενώ αν δεν ισχύει θα πρέπει να περιμένει μέχρι να μειωθούν τα παιδιά ή να μπει νέος δάσκαλος. Προφανώς αν η παραπάνω σχέση δεν ισχύει δηλαδή αν δεν μπορεί να φύγει ο δάσκαλος υπάρχει πιθανότητα να υπάρχουν παιδιά έξω από το νηπιαγωγείο τα οποία επιθυμούν να μουν μέσα. Προφανώς όσο υπάρχουν κενές θέσεις τα παιδιά είναι πιθανό να μπαίνουν παρά το γεγονός ότι ο δάσκαλος περιμένει να βγει. Αυτό οφείλεται στο γεγονός ότι χρησιμοποιούμε την ίδια μεταβλητή κατάστασης (condition variable) και για τα παιδιά και για τους δασκάλους, πράγμα το οποίο προσδίδει μια τυχαιότητα ως προς ποιά από τις διεργασίες που είναι σε αναμονή θα ξυπνήσει και θα προλάβει να μπει στο κρίσιμο τμήμα και να κλειδώσει. Συμπερασματικά, αν μπορεί να βγει ένας δάσκαλος αλλά και να μπει παιδί είναι τυχαίο ποιο από τα δυο θα συμβεί αλλά είναι σίγουρο πως μόνο μια διεργασία θα μπει στο κρίσιμο τμήμα, δηλαδή μόνο ένα από τα δυο αιτήματα θα ικανοποιηθεί, γεγονός το οποίο μας πιστοποιεί πως δεν θα βρεθούμε σε κάποια κατάσταση που είναι απαγορευμένη.

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.

Προφανώς στο κώδικα του αρχείου kgarten.c υπάρχουν race conditions οι οποίες τελικά επιλύθηκαν με αποτελεσματικό τρόπο ώστε να προκύψει σωστό αποτέλεσμα. Συγκεκριμένα τα race conditions είναι τα παρακάτω :

- Όταν μπει ή βγει ένα παιδί ή δάσκαλος καλείται η συνάρτηση `verify()` η οποία ελέγχει αν η προβλεπόμενη αναλογία μαθητών-δασκάλων τηρείται και αν δεν τηρείται εμφανίζει ένα μήνυμα λάθους και το πρόγραμμα τερματίζει. Πριν καλέσουμε την συνάρτηση αυτή γίνεται ένα `lock` και μόλις επιστρέψει η συνάρτηση γίνεται το αντίστοιχο `unlock`. Αυτό γίνεται διότι αν ένα νήμα καλούσε την `verify()` και εκείνη την ώρα ένα άλλο νήμα τροποποιούσε κάποιο από τα πεδία `vt-vc` ή καλούσε ξανά την `verify()` τότε η συνάρτηση αυτή θα έδινε λάθος αποτέλεσμα. Για να γίνει το `lock` πριν την εκτέλεση της `verify()` όπως και πριν την αυξομείωση των μεταβλητών `vt-vc` χρησιμοποιούμε ένα `mutex` και πλέον έχουμε αντιμετωπίσει την race condition αποτελεσματικά, αφού όταν ο `mutex` έχει κλειδώσει, κάποιο νήμα που ενδεχομένως θα προσπαθήσει να τροποποιήσει κάποια από τις μεταβλητές, θα πρέπει να περιμένει μέχρι να "ξεκλειδώσει" ο `mutex`, δηλαδή με τη χρήση του `mutex` διασφαλίζεται ότι κάθε φορά που ένα νήμα εκτελεί εντολές του κρίσιμου τμήματος οποιοδήποτε άλλο νήμα δε θα μπορέσει να εισέλθει στο κρίσιμο τμήμα.

- Άλλη μια race condition έχουμε όταν πληρούνται οι προϋποθέσεις ώστε να μπορεί να μπει παιδί αλλά και να βγει δάσκαλος. Για να καταφέρουμε να επιλύσουμε αποτελεσματικά κι αυτή ώστε να έχουμε πάντα σωστή αναλογία παιδιών και δασκάλων κάνουμε χρήση μιας μεταβλητής κατάστασης(condition variable) η οποία είναι κοινή για τα παιδιά και τους δασκάλους. Επομένως αν ένα παιδί επιθυμεί να μπει ή ένας δάσκαλος επιθυμεί να βγει και δεν μπορεί να ικανοποιηθεί καμία απο τις δυο ενέργειες, τότε εκτελείται η εντολή `pthread_cond_wait(&(thr->kg->condvar), &(thr->kg->mutex))` η οποία ξεκλειδώνει το mutex ώστε να μπορεί να χρησιμοποιηθεί απο κάποιο άλλο νήμα και το συγκεκριμένο νήμα θα πάει σε κατάσταση waiting και θα περιμένει κάποια αλλαγή. Αν τώρα ενα παιδί ή ένας δάσκαλος μπει στο νηπιαγωγείο τότε θα εκτελεστεί η εντολή `pthread_cond_broadcast(&thr->kg->condvar)` η οποία αυτο που κάνει είναι να στείλει σήμα “εκκίνησης” σε όλα τα νήματα που ήταν waiting, δηλαδή ξυπνάει όλα τα νήματα που είναι σε κατάσταση waiting. Ωστόσο, επειδή χρησιμοποιούμε μόνο ενα mutex το νήμα το οποίο θα προλάβει και θα μπει στο κρίσιμο τμήμα, θα κλειδώσει το mutex και αυτό θα έχει ως αποτέλεσμα τα υπόλοιπα νήματα να μεταβούν εκ νέου σε waiting και να περιμένουν ξανά κάποια αλλαγή. Θα πρέπει να τονιστεί οτι στις περιπτώσεις όπου γίνεται η εισαγωγή παιδιού και η έξοδος δασκαλού είναι αναγκαία η χρήση μιας while (δες στο κώδικα) αφού αν χρησιμοποιούσαμε if αντι για while θα μπορούσε μια αλλαγή να μην κάνει true την συνθήκη της if αλλά επειδή η συνθήκη της if ελέγχεται μόνο μια φορά δεν μπορεί να γίνει έλεγχος ξανά και είναι πολύ πιθανόν(σχεδόν βέβαιο) να οδηγηθούμε σε απαγορευμένη κατάσταση. Ωστόσο με την χρήση της while όταν ένα νήμα είναι σε κατάσταση waiting και σταλεί ενα `pthread_cond_broadcast(&thr->kg->condvar)` τότε θα τσεκάρει αν ισχύει η συνθήκη της while και ανάλογα το αποτέλεσμα το νήμα θα μεταβεί πάλι σε κατάσταση waiting ή θα συνεχίσει και θα εκτελέσει την αντίστοιχη ενέργεια, η οποία πλέον είμαστε σίγουροι πως δεν θα μας οδηγήσει σε απαγορευμένη κατάσταση.

Με τις παραπάνω ενέργειες είμαστε πλέον σίγουροι οτι αν και υπάρχουν race conditions αυτές σε κάθε περίπτωση επιλύονται αποτελεσματικά και δεν θα οδηγηθούμε ποτέ σε απαγορευμένη κατάσταση.